

APUNTES DE PASCAL

Departament de Matemàtica Aplicada
i Telemàtica

**Secció
Enginyeria
Telemàtica**

En la tabla inferior podemos ver la evolución del lenguaje:

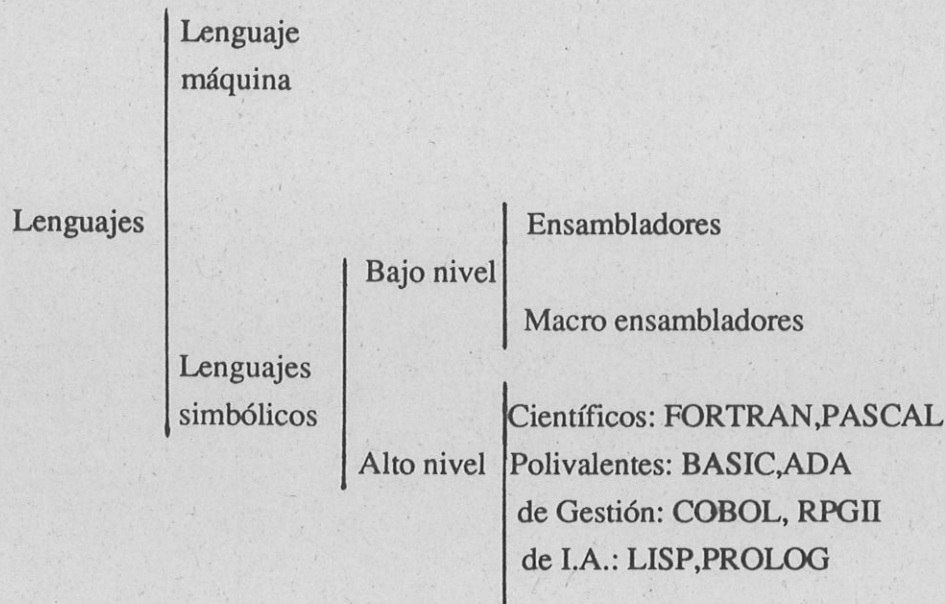
Inglés	Ensamblador (nemónico)	Código máquina
LOAD	LDA	10101001

Un programa escrito en lenguaje ensamblador, está formado por símbolos pertenecientes a un alfabeto que no es el binario, por tanto no pueden ser interpretados directamente por la máquina. Debido a ésto, es necesario traducir estos símbolos a un código máquina, para lo cual se utilizan los programas ensambladores.

El principal problema de la programación en ensamblador es que cada sentencia en ensamblador, equivale a una sola instrucción en código máquina, lo cual obligaba al programador a pensar en términos de instrucciones máquina individuales. Un primer paso para superar esta dificultad fue la creación de un lenguaje de programación cuyas sentencias representaran a más de una instrucción en código máquina, llamado macroensamblador. Aún así, la potencia de estos lenguajes era muy baja, por lo cual se crearon los lenguajes de alto nivel.

Estos lenguajes de programación de alto nivel, son más cercanos a los lenguajes naturales, teniendo así menos limitaciones para el programador que el lenguaje máquina. En estos lenguajes, al igual que en los ensambladores, se precisa de programas que nos traduzcan sus sentencias a instrucciones en código máquina. De esta forma, estos programas en lenguajes de alto nivel (PASCAL, COBOL, FORTRAN), se traducen a código máquina, mediante un traductor llamado compilador. Así, mediante un programa compilador adecuado, podremos ejecutar cualquier programa escrito en un lenguaje de alto nivel, ya que estos lenguajes están estandarizados.

Llamamos programa fuente a todo programa escrito en un lenguaje de alto nivel. Al resultado de la compilación de este programa fuente le llamaremos programa objeto, el cual tras un proceso de montaje o linkado (consultas a librerías del sistema, etc.), generará el programa que realmente podremos ejecutar en un computador.



Diremos que un sistema operativo, es un conjunto de programas que gestionan todos los recursos de la computadora, como introducir un programa, llamar al compilador, ejecutar el programa objeto resultante o realizar cualquier otra orden del sistema.

Llamaremos editor a un programa interactivo que se utiliza para crear y modificar los programas fuente o los datos.

LENGUAJE PASCAL

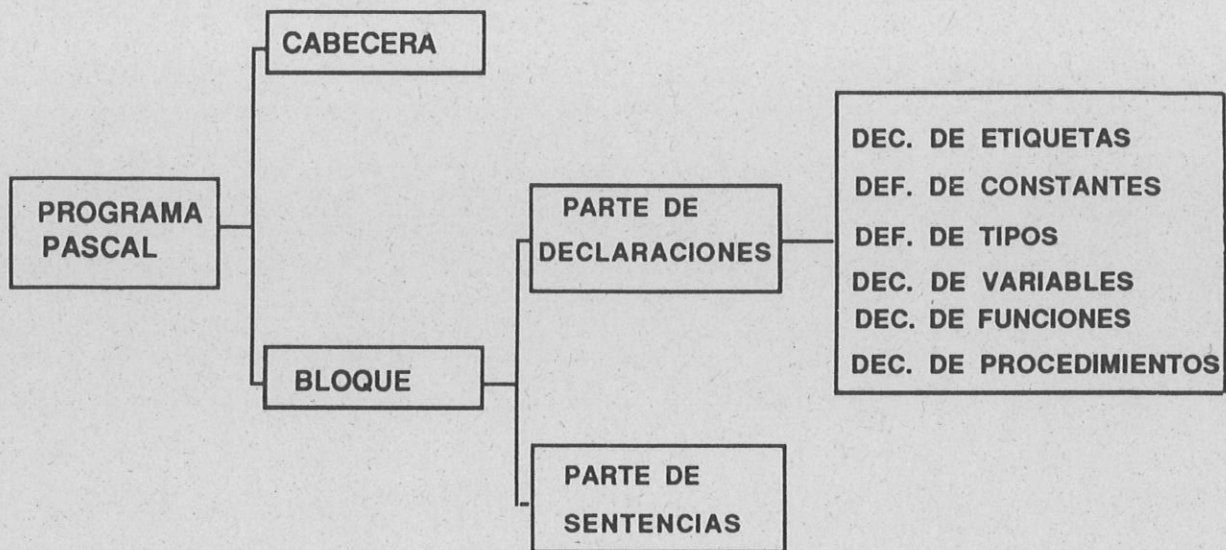
2.1. Introducción.

El lenguaje de programación PASCAL fue diseñado por un científico informático suizo llamado Niklaus Wirth en 1968 y revisado en 1972. El nombre del lenguaje se debe al famoso matemático francés Blaise Pascal quien diseñó una de las primeras máquinas de calcular mecánicas.

Las principales características de este lenguaje son:

- Lenguaje compilado.
- Lenguaje estructurado: se presta bien a las técnicas de programación modular y estructurada propias del análisis descendente.
- Permite la definición de tipos de datos " a medida " según la necesidad del programa.
- Permite el uso de estructuras recursivas, mediante las cuales podemos acceder a subprogramas que se llaman a "ellos mismos".
- Facilita el adecuado aprendizaje de las técnicas de programación .

2.1.1. Estructura general de un programa PASCAL.



2.1.2. Identificadores. Diagramas sintácticos.

Identificadores:

Los identificadores son utilizados en Pascal para denominar las cosas. Algunos han sido definidos en el lenguaje y se reservan para usos específicos. De esta forma todo nombre no incluido en los reservados podrá ser definido por el usuario para su propio uso. Por tanto:

Identificador es todo nombre asociado con procesos y objetos que se utiliza para referenciarlos.

Los identificadores se forman con letras (A-Z,a-z) y números (0-9) comenzando siempre con una letra.

* Ejemplo:

validos	no validos
ZOR88	ZOR.88
K20	20K
XY	X-Y

La longitud de los identificadores no está limitada aunque algunos compiladores sólo reconocen un número fijo de caracteres del identificador.

La computadora funcionará igual sea cual sea el nombre que utilicemos para el identificador, siempre y cuando usemos siempre el mismo para definir a un mismo

proceso. En cambio para el programador, si el identificador tiene un significado referente al proceso le será mucho más fácil imaginar como funcionará el programa. Por tanto siempre que queramos dar un nombre a algo en un programa, debe elegirse de tal forma que tenga sentido para un usuario.

Las letras mayúsculas y minúsculas se tratarán como si fueran iguales.

Diagramas sintácticos:

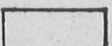
Entendemos como:

Sintaxis: Reglas formales que gobiernan la construcción de sentencias válidas en un lenguaje.

Semántica: Conjunto de reglas que dan el significado de una sentencia del lenguaje.

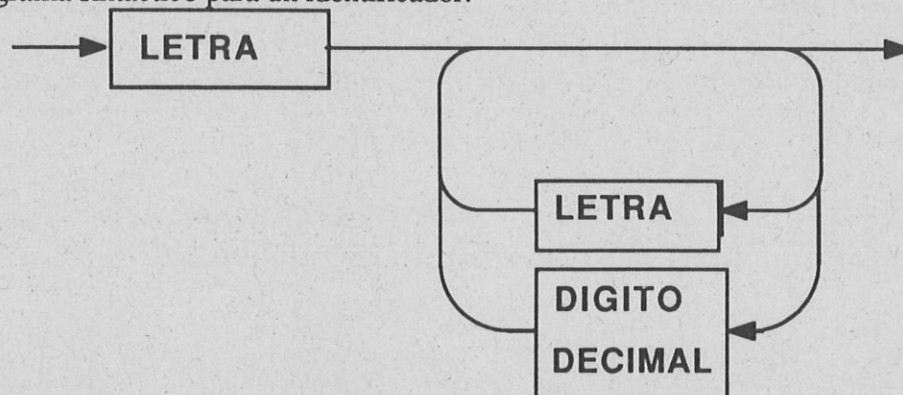
Los diagramas sintácticos son una descripción precisa de la forma correcta de construir programas y sus sentencias:

Características de un diagrama sintáctico:

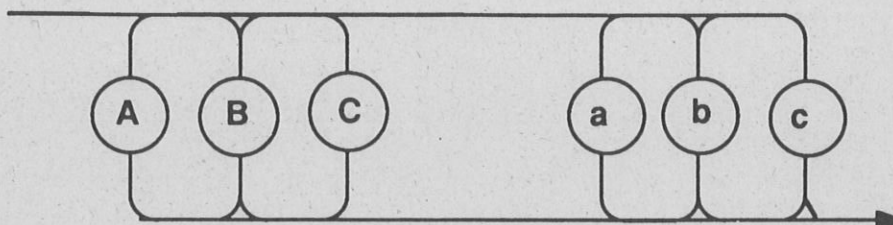
- Se indica con ---> el camino a seguir por el diagrama. Existirá un solo inicio y un solo final.
- Cada bloque  encontrado en el camino se, sustituye por la definición del mismo dada por otro diagrama sintáctico.
- Cada bloque  encontrado en el camino se sustituye por su contenido literal.
- Cada bifurcación del camino  permite optar por cualquiera de los caminos que se indican, eligiendo el camino que en cada caso convenga.
- En el diagrama sólo será necesario indicar el sentido del camino con una flecha en la entrada de cada bloque y en el final del diagrama.

* Ejemplo:

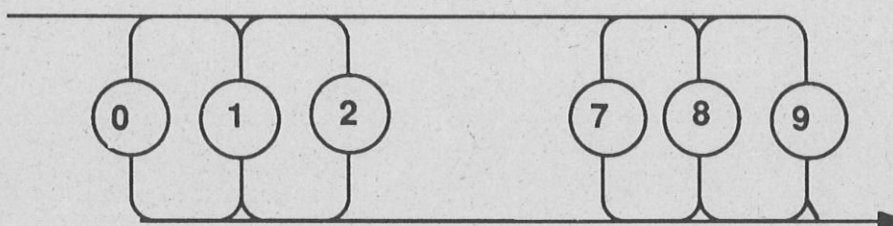
Diagrama sintáctico para un identificador:



Letra:



dígito decimal:

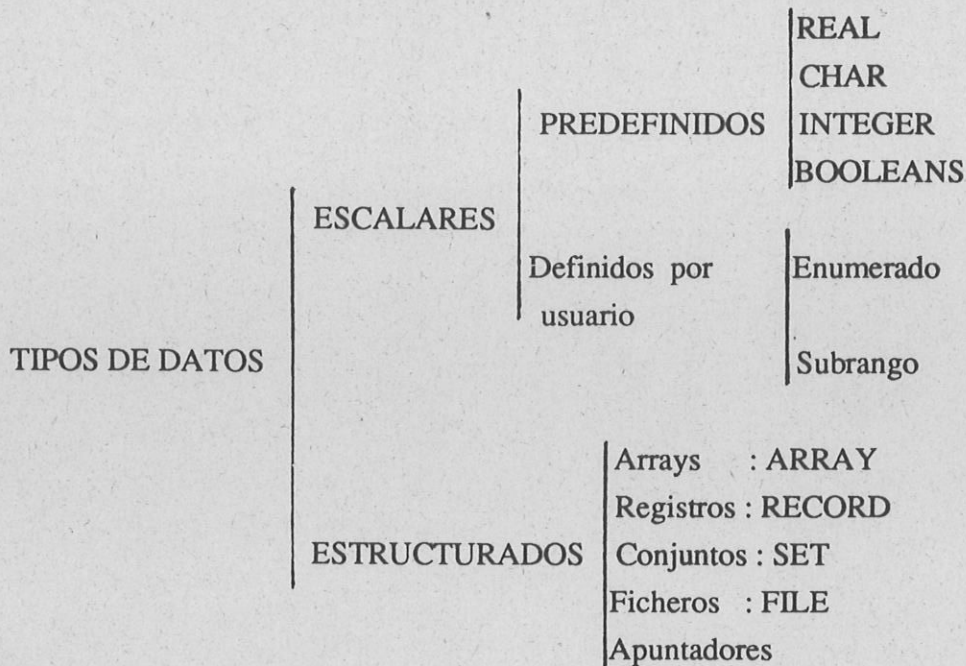


2.2. Datos: tipos, constantes y variables.

Cuando escribimos un programa, los datos son los elementos que suministramos al mismo, los cuales deben traducirse a una forma más útil. En PASCAL cada elemento de datos debe ser de un tipo específico. El *tipo de datos* determinará el tratamiento que van a recibir por parte de la computadora, o sea la manera en como van a ser procesados. Esta afirmación nos obliga a definir cada tipo de dato al inicio de cada programa, para que la computadora sepa en cada momento cómo el dato en cuestión ha de ser tratado dentro del programa.

2.2.1. Tipos de datos.

Una clasificación de tipos de datos la podemos ver en el gráfico de la página siguiente.



1.- INTEGER:

Los enteros son todos los números positivos o negativos (sin parte fraccionaria). Constan de un signo y dígitos, de tal forma que cuando se omite el signo se toma al número como positivo. No se permiten las comas.

Existe una limitación debida al hardware de la computadora que nos obliga a establecer un rango de valores para los números enteros. Como este límite varía de unas máquinas a otras el lenguaje PASCAL utiliza un identificador predefinido, *MaxInt*.

MaxInt : devuelve el entero positivo mayor.

Por ejemplo podemos ver los valores tomados para dos versiones de PASCAL:

$$MaxInt = 2E31 - 1 = 2.147481247 \text{ (VAX PASCAL)}$$
$$MaxInt = 2E15 - 1 = 32767 \text{ (TURBO PASCAL).}$$

De esta forma vemos que para el TURBO PASCAL el rango de valores estará entre:

- <i>MaxInt</i>	hasta	<i>MaxInt</i>
- 32767	hasta	32767

* Ejemplo:

válidos: 4 +22 -250 3 0

no válidos: 4. -22.0

Otros tipos de INTEGER son:

Para el VAX:

UNSIGNED : 0,1,...,(2E32-1)

Para el TURBO PASCAL:

BYTE : 0.....255 (8 bits)

SHORTINT : -128.....127 (8 bits)

LONGINT : -2E31.....+(2E31-1)

INTEGER : -2E15.....+(2E15-1) (16 bits)

WORD : 0.....(2E16-1) (16 bits)

2.- REAL:

Los números reales son números decimales, o sea tienen una parte entera y una parte fraccionaria con un punto decimal entre ellas. Pueden tener un exponente, el cual se expresará detrás de una 'E'.

Cuando utilicemos números reales en un programa, hemos de introducir un dígito, como mínimo, a cada lado del punto decimal.

* Ejemplo:

válido	no válido
3.1415	.42
0.43	15.
-1.0	.0
0.0	0.

En el esquema inferior podemos ver una representación de los valores de la precisión para distintos rangos

VAX PASCAL:

REAL

	<u>SINGLE</u>	<u>DOUBLE</u>	<u>DOUBLE</u>	<u>QUADRUPLE</u>
	(F_FLOATING)	(D_FLOATING)	(G_FLOATING)	(H_FLOATING)
	-1.70E38	-1.70D38	-0.90D308	-0.59Q4932
	-0.29E-38	-0.29D-38	-0.56D-308	-0.84Q-4932
	0.29E-38	0.29D-38	0.56D-308	0.84Q-4932
	1.70E38	1.70D38	0.90D308	0.59Q4932
díg.dec	7	16	15	33
bytes	4	8	8	16

TURBO PASCAL:

<u>REAL</u>	<u>SINGLE</u>	<u>DOUBLE</u>	<u>EXTENDED</u>
--10E38	--10E38	--10E307	--10E-4931
--10E-38	--10E-38	--10E-307	--10E4931
díg.dec. 11	6-7	16-17	19
bytes 6	4	8	10

3.- CHAR:

En este tipo de datos describiremos los que son un caracter alfanumérico, en los cuales van a estar incluidos letras, dígitos y símbolos especiales.

*Ejemplo:

'A' 'a' '8' '+' '-' '\$' '&' '*'

Como vemos en el ejemplo anterior, cualquier caracter alfanumérico puede representarse entre comillas simples como un caracter CHAR, si bien hemos de tener en cuenta que la misma comilla simple no podrá ser introducida entre comillas como un caracter CHAR.

El hecho de utilizar las comillas es debido a que el compilador PASCAL las precisa para diferenciar, por ejemplo, entre el dato caracter '8' y el valor entero 8. Observamos que el blanco (espacio en blanco) es también un caracter ' '.

Cada máquina tiene un conjunto de caracteres que pueden ser representados en ella, llamado conjunto de caracteres de la máquina.

Dos valores CHAR no van a poder ser sumados pero en cambio si que los vamos a poder comparar.

*Ejemplo:

'1' es menor que '2' 'b' es menor que 'c'

El conjunto de caracteres se encuentran tabulados: código ASCII (ver apéndice). Presentan una ordenación tal que, las letras están colocadas en orden alfabético y los números en orden creciente del 0 al 9.

4.- BOOLEAN:

Es un tipo de dato con sólo dos valores: *True* y *False* (verdadero y falso). Se utilizan para representar la respuesta a preguntas.

5.- DATOS DEFINIDOS POR USUARIO

Para la identificación de un tipo de dato en el programa utilizaremos la palabra reservada TYPE. Las características de este tipo de datos se verán en el apartado 2.8 (Pag 47)

2.2.2. Constantes.

Todos los números enteros y reales, son constantes. Además también existen caracteres y series de caracteres llamados cadenas.

* Ejemplo:

25 45.4 'b' 'what is up'

Estas constantes podrán ser utilizadas en el programa, siempre que sea apropiado como parte de las expresiones.

Llamaremos valor literal o literal a cualquier valor constante establecido en el programa.

En el ejemplo anterior vemos que los literales caracteres y los literales cadena están entre comillas simples de manera que el compilador pueda diferenciar entre cadenas e identificadores.

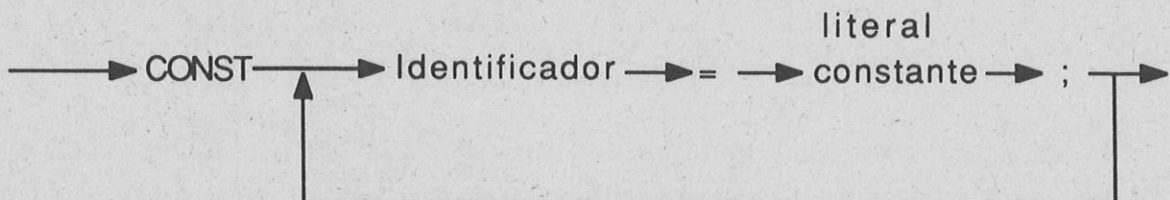
* Ejemplo:

'oper' es la cadena de caracteres formada por las letras *o,p,e,r*
oper es un identificador.

Los identificadores deben comenzar por una letra y las constantes enteras o reales empiezan con un dígito o con un signo +, -. De esta forma no es necesario colocar comillas para diferenciarlos

Cuando una constante ha de aparecer varias veces en un programa se le asigna un nombre, el cual será utilizado a lo largo del programa. Tales constantes se llaman *constantes con nombre*, y se definen en la sección de declaraciones del programa.

En el siguiente gráfico podemos ver un diagrama sintáctico, el cual nos muestra como expresar una constante.



La palabra reservada CONST, denota las declaraciones de constantes.

Se trata de una palabra reservada puesto que tiene un significado especial en el lenguaje PASCAL, por lo cual no podremos utilizarla como identificador.

2.2.3. Variables.

Todo programa opera sobre datos los cuales pueden ser de uno de los tipos simples que hemos visto en el apartado anterior: Integer, Real, Boolean y Char.

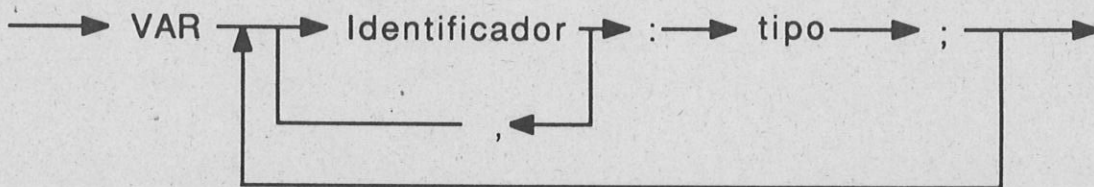
Estos datos van a ser almacenados en posiciones de memoria. El nombre simbólico asignado a esta posición de memoria (variable) es el nombre de la variable o identificador de la variable.

Sin embargo, en la práctica el nombre de la variable es referido como la variable y su valor es lo que cambia.

Las variables deben ser declaradas al igual que las constantes y los tipos de datos en la sección de declaración de un programa PASCAL. Así, todos los identificadores en PASCAL son declarados o definidos antes de ser usados. De ahí que la sección de declaración esté antes que el bloque de un programa.

Cuando el compilador traduzca el programa fuente al objeto, verificará que no se escriban instrucciones que intenten almacenar un valor de un tipo de dato erróneo. Si tras la comprobación el compilador detecta algún error de este tipo, dará un mensaje de error.

En el gráfico siguiente podemos observar un diagrama sintáctico de una declaración de variable.

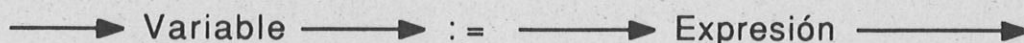


La palabra reservada VAR denota las declaraciones de variables.

En estos instantes hemos definido tres palabras reservadas para la sección de declaraciones, que son: CONST, TYPE, VAR. La sección de declaraciones, irá estructurada en este orden.

2.2.4. Sentencia de asignación.

Para cambiar el valor de una variable utilizaremos una sentencia de asignación, cuya construcción podremos ver en el siguiente diagrama sintáctico:



El operador de asignación es ":", y se podría leer como "toma"; la variable toma el valor de la expresión. Sólo puede haber una variable en el lado izquierdo de la expresión, la cual ha de ser del mismo tipo que el valor que se le va a asociar. Una variable puede tomar distintos valores en un programa, siempre y cuando cada resultado sea cambiado por otra sentencia de asignación.

2.3. Expresiones.

2.3.1. Expresiones aritméticas. Operadores.

Las expresiones pueden estar formadas por variables, constantes y operadores. Los operadores aritméticos son:

+	suma
-	resta
*	multiplicación
/	división
DIV	división entera
MOD	módulo(resto de DIV)

Las dos últimas sentencias sólo se utilizan para dividir números enteros, puesto que para dividir los reales ya tenemos el operador división.

Si un operando es real y el otro es entero, se convierte el valor entero en real antes de que se ejecute la operación.. Si se utiliza la / entre dos operandos enteros, los dos operandos se convierten a real antes de que se haga la operación. La conversión de los valores enteros a reales se hace sólo temporalmente con el propósito de una división particular, y la conversión no afecta a los valores que están realmente almacenados en las variables enteras.

* Ejemplo:

Dada la declaración de variables siguiente:

VAR

x,y: Integer;

z:Real;

Ch:Char;

Comparación:Boolean;

Podemos escribir las siguientes asignaciones:

```
x:=2;
y:=15;
z:=0.23;
Ch:='w';
Comparación:=False;
```

Operador	Real Real	Real Integer	Integer Real	Integer Integer
+	Real	Real	Real	Integer
-	Real	Real	Real	Integer
*	Real	Real	Real	Integer
/	Real	Real	Real	Real
MOD	Error	Error	Error	Integer
DIV	Error	Error	Error	Integer

En el cuadro anterior se puede ver cuál será el tipo de resultado dados los operadores y el operando.

2.3.2. Expresiones Booleanas. Operadores relacionales y Booleanos.

Sabemos que el tipo de datos Boolean tiene sólo los dos valores *True* o *False*. Diremos que una expresión es Booleana cuando sus posibles resultados sólo pueden ser datos Booleanos.

Una expresión Booleana puede ser:

1.- Variable Booleana. Es aquella que está declarada de tipo Boolean: su contenido puede ser True o False. Se distingue de las variables declaradas del tipo Integer, Real o Char en que sus valores no pueden ser leídos, pero sí escritos.

2.- Una expresión más un operador relacional y otra expresión. Podemos asignar valores a las variables Booleanas igualándolos al resultado de una expresión más un operador relacional y otra expresión. Al comparar dos valores nos aparece una relación entre ellos. Las relaciones que podemos realizar en PASCAL son:

=	igual a	>=	mayor o igual que
<>	no igual	>	mayor que
<=	menor o igual que	<	menor que

Estos símbolos reciben el nombre de *operadores relacionales*, pues hacen que la computadora ejecute operaciones que verifican si la relación existe o no entre dos valores. Siempre hemos de comprobar que los dos terminos a comparar sean del mismo tipo (número o caracteres). También podremos utilizar operadores relacionales para comparar directamente el resultado de expresiones aritméticas.

3.- Expresión Booleana más operador Booleano y otra expresión Booleana. Diremos que los *operadores Booleanos* son los símbolos especiales AND, OR y NOT, sólo definibles para expresiones Booleanas. Podemos combinar operadores relacionales con operadores Booleanos.

* Ejemplo:

$(A > B) \text{ AND } (A > C)$

Esta relación sólo será True (verdadera) si se cumple que A es mayor que B y A es mayor que C.

La operación OR coge los valores Booleanos y los combina de forma que si uno de ellos o ambos son True el resultado será True.

Las siguientes tablas resumen los valores que se obtendrían al aplicar los operadores Booleanos a un par de valores Booleanos (X,Y).

<u>VALOR DE X</u>	<u>VALOR DE Y</u>	<u>X AND Y</u>	<u>X OR Y</u>	<u>NOT X</u>
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

2.3.3. Prioridad de los operadores.

En el siguiente dibujo, podemos ver el orden de prioridad de los operadores aritméticos, relacionales y Booleanos:

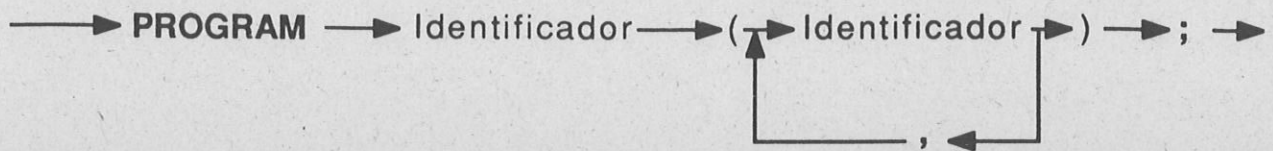
OPERADORES	PRECEDENCIA
NOT	Mayor
* *	
*,/,DIV,MOD,AND	
+, -, OR	
=, <, >, <=, >=	
	Menor

Los operadores de la misma línea tienen la misma precedencia. Si en una expresión nos encontramos con más de un operador de igual precedencia, la computadora los evalúa de izquierda a derecha. Para variar esta regla pueden usarse paréntesis, los cuales utilizaremos siempre en caso de duda. El compilador quitará los paréntesis innecesarios. A la hora de utilizar los paréntesis siempre los colocaremos por pares.

2.4. Sentencias de entrada y salida de datos.

2.4.1. Cabecera del programa. Ficheros INPUT y OUTPUT.

En el siguiente diagrama sintáctico podemos ver la construcción de un cabecera de programa:



Podemos ver que la cabecera está compuesta de la palabra reservada **PROGRAM** seguida de un identificador que será definido por el usuario y una lista de los ficheros usados en el programa los cuales nos permiten la comunicación entre el programa y el mundo exterior. INPUT es normalmente el teclado del terminal o el periférico por el cual se entran datos y OUTPUT la pantalla del terminal o la impresora.

2.4.2. Sentencias de entrada de datos.

El proceso de introducir valores desde un conjunto de datos exteriores en las variables de programa se llama entrada.

En PASCAL, este proceso se realiza mediante la sentencia READ.

En la construcción de una sentencia con Read, hemos de vigilar en introducir sólo variables en la lista de parámetros, las cuales de haber varias, estarán separadas por comas.

Para separar varios números, en la entrada de datos por teclado, utilizaremos el espacio en blanco.

Al igual que en la sentencia de asignación, se ha de evitar introducir valores reales en una variable declarada del tipo entero. Sin embargo, podremos leer un valor entero en una variable real.

Para ordenar la asignación de valores con las variables, la primera variable de la lista de parámetros de la primera sentencia Read del programa obtendrá el primer número que se reciba, la segunda variable el segundo valor y así sucesivamente. Si se escriben más números que variables hayan en la lista de parámetros, la computadora almacenará los números que le sobren hasta que encuentre la siguiente sentencia Read. En el caso de no existir ninguna más, la computadora los despreciaría. Si se introducen menos datos que variables existen en la sentencia READ, el programa queda a la espera de nuevos datos hasta poder completar los valores de sus variables.

En el siguiente cuadro se pueden ver distintas formas de construir una sentencia Read.

<u>SENTENCIA</u>	<u>DATOS</u>	<u>CONTENIDAS DESPUES LECTURA</u>
Read (X)	47	X = 47
Read (X, Y, Z)	4 7 15	X = 4, Y = 7, Z = 15
Read (X, Y)	47	
	26	X = 47, Y = 26
Read (X, Y)	44 32.8	X = 44, Y = 32.8

Otra sentencia utilizada en PASCAL para la introducción de datos es READLN.

Su diferencia con respecto a Read es que una vez las variables de la lista de parámetros han recibido su valor correspondiente se salta a la siguiente línea, obviando todo lo que queda en la que estaba leyendo.

- *Indicador marcador de lectura y <eoln>*:

El indicador marca un lugar en la línea de entrada, de forma que señala el siguiente carácter que aún no ha sido leído. O sea nos marca el punto de la línea de valores de entrada por el cual la computadora seguirá leyendo.

Cada línea de entrada tiene un carácter fin de línea <eoln> que advertirá a la computadora donde termina la línea y donde empieza la siguiente. Las sentencias Read y Readln cruzarán los límites de la línea <eoln> para encontrar tantos valores como variables tenga en su lista de parámetros.

El <eoln> es un carácter de control especial no imprimible que un sistema reconoce, de forma que es tratado como un espacio en blanco cuando una sentencia está leyendo valores.

Se genera mediante un RETURN o un WRITELN (se verá más adelante).

* Ejemplo:

Dadas las tres líneas de valores de entrada siguientes, vemos como variarán el marcador de línea y el <eoln>. En negrita se marca la parte de datos que han sido leídos después de la sentencia.

```

762      431      128<eoln>
256      348      950<eoln>
592      125      673<eoln>

```

A,B,C,D: Integer

<u>SENTENCIAS</u>	<u>VALORES LEIDOS</u>	<u>POSICION MARCADOR</u> <u>DESPUES Read/Readln</u>
Read (A, B);	A = 762, B = 431	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>
Read (C, D);	C = 128, D = 256	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>
Readln (A, B);	A = 762, B = 431	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>
Readln (C, D);	C = 256, D = 348	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>
Read (A, B);	A = 762, B = 431	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>
Readln (C, D);	C = 128, D = 256	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>
Readln (A, B);	A = 762, B = 431	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>

Readln (C);	C = 256	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>
Readln (D);	D = 592	762 431 128<eoln> 256 348 950<eoln> 592 125 673<eoln>

Los caracteres se tratan de forma diferente a los datos numéricos en una lectura. Una variable Char almacena sólo un caracter, con lo cual cuando el programa esté leyendo valores en variables de este tipo, sólo se leerá un caracter.

Después de un Readln el marcador de lectura estará al inicio de una línea. Después de un Read el marcador de lectura indica el siguiente valor que va a continuación del último que se acaba de leer. Esto puede ser un problema si se quiere que la computadora lea datos de caracteres seguidos de un número. Como un número generalmente va seguido de un espacio en blanco, y éste es un caracter, el espacio será leído en la variable Char.

Para solucionar este problema, podremos escoger entre dos opciones:

- Declarar otra variable de tipo Char, que no se va a utilizar en ningún cálculo del programa, pero que va a asignarse con el espacio en blanco no deseado.
- Leer el dato dos veces. La primera vez asignaremos a la variable Char el espacio en blanco, y a la segunda lectura el valor deseado será asignado a la variable Char, reemplazando al espacio en blanco.

* Ejemplo:

Dada la siguiente lista, aplicamos las siguientes sentencias:

```
56 12 AB
P,Q=Integer;
Ch1,Ch2=Char;
```

<u>SENTENCIAS</u>	<u>VALORES LEIDOS</u>	<u>POSICION MARCADOR DESPUES Read/Readln</u>
Read (P, Q);	P = 56, Q = 12	56 12 AB<eoln>
Read (Ch1);	Ch1 = ''	56 12 AB<eoln>
Read (Ch2);	Ch2 = 'A'	56 12 AB<eoln>

Como vemos el espacio en blanco es asignado a una variable Char, y de la manera como está hecha esta lectura de datos, nunca podremos llegar a leer los 4 datos con las 4 variables. Para solucionarlo, podemos hacer.

<u>SENTENCIAS</u>	<u>VALORES LEIDOS</u>	<u>POSICION MARCADOR</u> <u>DESPUES Read/Readln</u>
Read (P, Q);	P = 56, Q = 12	56 12 AB<eoln>
Read (Ch0);	Ch0 = ''	56 12 AB<eoln>
Read (Ch1);	Ch1 = A	56 12 AB<eoln>
Read (Ch2);	Ch2 = B	56 12 AB<eoln>

La opción usada en este caso ha sido la de incluir una nueva variable, que será la que nos leerá el espacio en blanco, y que deberemos definir en la sección de declaraciones al principio del programa. Otra posible solución es la siguiente

<u>SENTENCIAS</u>	<u>VALORES LEIDOS</u>	<u>POSICION MARCADOR</u> <u>DESPUES Read/Readln</u>
Read (P, Q);	P = 56, Q = 12	56 12 AB<eoln>
Read (Ch1);	Ch1 = ''	56 12 AB<eoln>
Read (Ch1);	Ch1 = A	56 12 AB<eoln>
Read (Ch2);	Ch2 = B	56 12 AB<eoln>

Se puede ver que en esta solución lo que se hace es asignar a la variable Ch1 dos valores, el primero no deseado el espacio en blanco, y después el valor B, valor deseado que será el que utilizará para los cálculos pertinentes del programa.

2.4.3. Sentencias de salida de datos.

La sentencia que nos permitirá escribir los resultados obtenidos de los cálculos de un programa será WRITE.

En la lista de parámetros introduciremos lo que queramos imprimir, siendo la función de la sentencia coger los datos e imprimirlos.

La sentencia WRITE es un subprograma cuya misión es que la computadora ponga temporalmente al programa en suspensión empezando a ejecutarse el subprograma Write, dándole los datos desde la lista de parámetros. Cuando éste ha finalizado y ya se ha imprimido todo lo deseado, el programa arranca de nuevo desde el punto donde había sido parado.

La sentencia Write es un subprograma del tipo procedimiento que veremos más adelante.

Como en la lista de parámetros de la sentencia Read, las expresiones deberán ir separadas por comillas.

* Ejemplo:

```
A:=2;
Write ('EL RESULTADO ES = ', A);
```

En pantalla:

EL RESULTADO ES = 2

Vemos que la sentencia Write imprime los caracteres que están entre comillas simples, o lo que es lo mismo imprime las constantes cadenas exactamente como aparecen en la lista de parámetros, pero sin las comillas que se utilizan para encerrarlas. Si quisiéramos escribir una cadena en la cual hubiese una comilla deberíamos poner en su lugar dos comillas simples sin espacio en blanco entre ellas.

Con la sentencia Write podemos definir el número de decimales a escribir (en caso de números reales), y el número de caracteres con los cuales formatearemos los datos usados.

De esta forma:

WRITE (A:E:D)

donde: - A es el dato a escribir.

- E es el número de espacios utilizados para representar al número.

- D es el número de decimales a escribir.

Si los valores de A son del tipo INTEGER sólo colocaremos el número de espacios utilizados para la representación del número.

* Ejemplo:

```
A:=2.346;
Write (A:8:2);
```

En pantalla

				2	.	3	5
--	--	--	--	---	---	---	---

Observamos que además de eliminar la última cifra nos redondea el resultado.

* Ejemplo:

A:=2.34;

Write (A:J:2);

En pantalla

J = 1-4

2	.	3	4		
---	---	---	---	--	--

J = 5

	2	.	3	4	
--	---	---	---	---	--

J = 6

		2	.	3	4
--	--	---	---	---	---

Como se ve en el ejemplo anterior, en el primer caso toma un mismo formato de 4 caracteres para los casos de J = 1 hasta 4, o sea cuando el formato que definamos para el número sea menor que el número de caracteres del dato, se tomará el mínimo número de caracteres necesarios para escribir el dato.

Otra sentencia que nos permite la salida de datos por periférico es la WRITELN., la cual escribe en pantalla el contenido de los paréntesis (variables o caracteres entre comillas) y coloca el cursor en la siguiente línea.

Veamos un ejemplo comparativo entre las dos sentencias de salida de datos.

WRITE ('SALTO DE LINEA');

WRITE ('SIGUIENTE');

Por pantalla:

SALTO DE LINEA SIGUIENTE

WRITELN ('SALTO DE LINEA');

WRITE ('SIGUIENTE');

Por pantalla:

SALTO DE LINEA <eoln>
SIGUIENTE

2.4.4. Entrada/Salida Interactiva.

Un programa interactivo es aquel en el que existe una comunicación entre la computadora y el usuario.

Cuando nosotros creamos un programa, generalmente es de naturaleza interactiva, por tanto debemos suministrar instrucciones claras y precisas que indiquen como han de ser los datos a introducir por el usuario. Para ello utilizaremos las sentencias WRITE o WRITELN con el mensaje adecuado, antes de la sentencia READ (lectura del dato introducido).

* Ejemplo:

```
WRITELN ('Introduzca un número entero: ');  
READ (n);
```

Por pantalla
Introduzca un número entero:
3

La cantidad de información que debe ponerse en los mensajes dependerá del usuario al que vaya destinado el programa, siendo ésta más clara y especificativa cuanto menor sea el conocimiento de la persona que va a utilizar el programa sobre el tema a tratar.

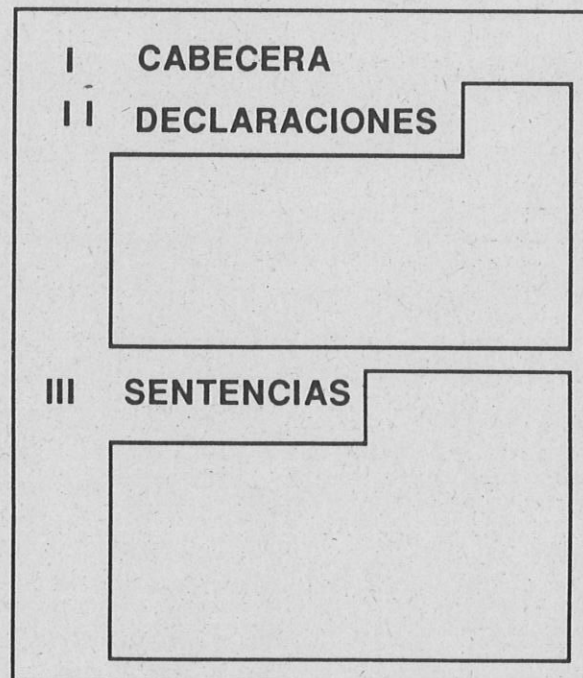
2.4.5. Construcción de programas.

Hasta ahora hemos visto los elementos básicos para la creación de un programa (tipos de datos, declaraciones, sentencias, entrada/salida de datos,...).

En este apartado vamos a ver de que forma los tenemos que colocar para que adquieran la forma de un programa en PASCAL.

La estructura de un programa en PASCAL está formada por tres bloques colocados como nos muestra la figura siguiente:

PROGRAMA PASCAL



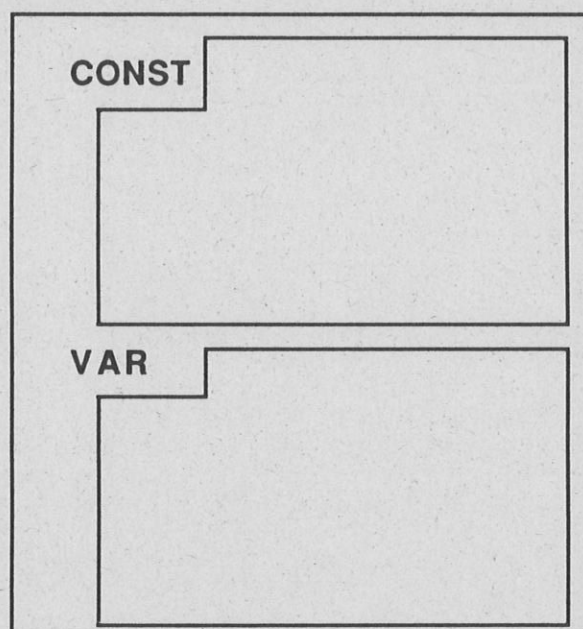
El primer paso a la hora de construir un programa será colocar la cabecera de la forma en que se especifica en el apartado 2.13.1.

A continuación nos encontramos con la sección de declaraciones. En ella definiremos las constantes que vamos a utilizar a lo largo del programa y las variables, indicando el tipo del dato que van a contener.

La construcción sintáctica de las constantes y variables se realizará tal y como indica el apartado 2.11. Las palabras reservadas CONST y VAR sólo aparecen una vez al principio del conjunto de constantes y de variables respectivamente.

El orden de colocación es el que se muestra en la figura inferior.

DECLARACIONES



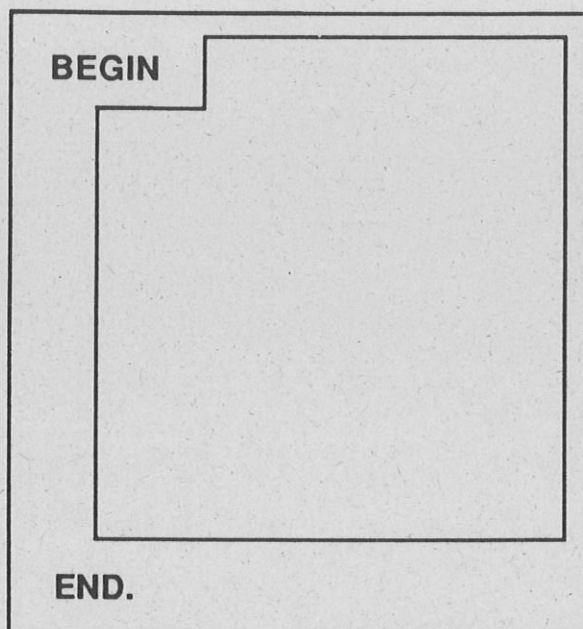
Por último tenemos la sección de sentencias. En ella colocaremos las instrucciones que serán traducidas en la fase de compilación y que formarán la parte ejecutable del programa.

Esta sección va colocada entre un par BEGIN - END tal y como se indica en la figura de la página siguiente.

La palabra BEGIN indica el principio de la sección de sentencias y END. (con el punto final), indica donde acaba dicha sección. Todo aquello que se encuentre fuera de estas dos palabras no será interpretado por el compilador como parte de la sección de sentencias, también llamada programa principal.

Dentro de esta sección cada sentencia irá separada por un punto y coma (;), excepto la que se encuentre delante de un END, en cuyo caso no es necesario colocarlo.

SENTENCIAS



El par BEGIN - END también se utiliza en las sentencias compuestas (o conjunto de sentencias) como elemento de agrupación.

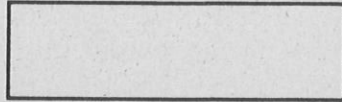
Por tanto, un programa en PASCAL tendrá la estructura que se muestra en la página siguiente.

Hemos de señalar que esta estructura de programa no es la más completa pero si es suficiente para hacer funcionar un programa. Si nuestro planteamiento de problema no contempla la aparición de constantes o variables podemos omitir cualquiera de los dos apartados o ambos.

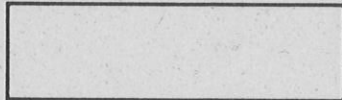
Más adelante iremos completando esta estructura con la inclusión de nuevos elementos en la sección de declaraciones (procedimientos, funciones,...) y nuevas sentencias y estructuras en la sección de sentencias (WHILE, REPEAT, IF,...). Por el momento tenemos suficiente como para ver un ejemplo de programa en PASCAL.

PROGRAM Identificador **(INPUT,OUTPUT);**

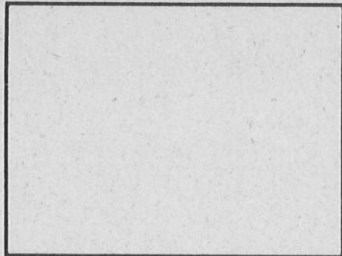
CONST



VAR



BEGIN



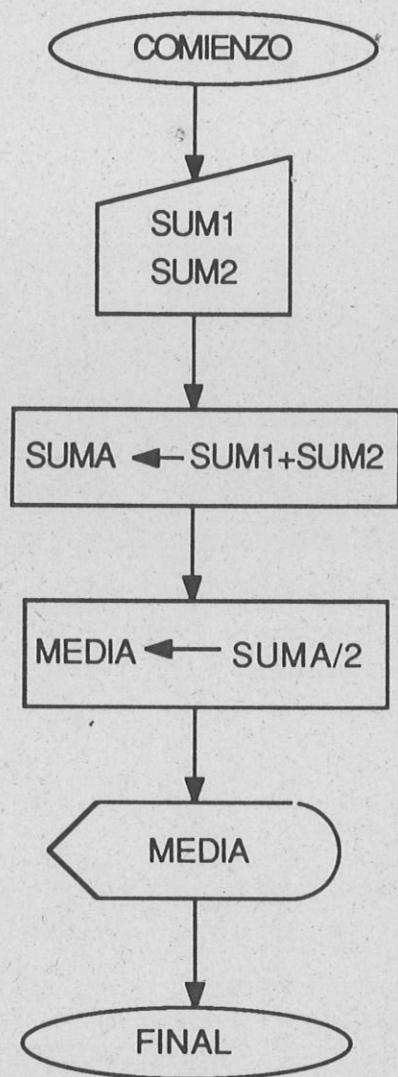
END.

* Ejemplo:

Calcular la media aritmética entre dos valores enteros cualesquiera. La introducción de datos se realizará por teclado y el resultado saldrá por pantalla. Este estará compuesto por una parte entera seguida de un solo decimal si la operación precisa de números decimales. Construir su diagrama de flujo y el programa en lenguaje PASCAL.

La primera acción para realizar el programa en lenguaje PASCAL, será resolver el problema planteado. Para ello construiremos el algoritmo. A la vez, hallaremos el pseudocódigo, del cual traduciremos al lenguaje de programación deseado, PASCAL.

En la página siguiente se puede ver el algoritmo que nos resolvería el problema.



PSEUDOCODIGO

```

BEGIN
  READ (SUM1);
  READ (SUM2);
  SUMA:=SUM1+SUM2;
  MEDIA:=SUMA/2;
  WRITE ('MEDIA')
END.
    
```

A continuación podemos observar la estructura del programa definitivo, realizado en lenguaje PASCAL.

```
PROGRAM Aritmetica (INPUT, OUTPUT) *
```

```
CONST
```

```
    PAR = 2 ;
```

```
VAR
```

```
    SUM1,SUM2,SUMA: INTEGER;
    MEDIA:REAL;
```

```
BEGIN
```

```
    WRITELN ('Introduce dos números enteros');
    WRITE ('Número 1= ');
    READ (SUM1);
    WRITE ('Número 2= ');
    READ (SUM2);
    SUMA:=SUM1+SUM2;
    MEDIA:=SUMA/PAR;
    WRITELN
    WRITE ('La media aritmética entre',SUM1:2,' y ',
           SUM2:2,' es: ',MEDIA:5:1)
```

```
END.
```

2.5. Estructuras condicionales.

Hasta ahora en la estructura secuencial tan solo hemos permitido que el programa siga un camino. Ahora bien, puede ser que en un momento dado se nos plantee un camino alternativo en función de un determinado suceso. O sea ¿qué sucede si lo que queremos es tomar una decisión dentro del programa?.

Este problema lo solucionaremos mediante las estructuras condicionales o alternativas.

Las estructuras condicionales o estructuras de control de selección permiten que la computadora escoja una de entre varias sentencias para ejecutarlas.

2.5.1. Sentencia IF - THEN - ELSE.

Para definir estas estructuras el lenguaje PASCAL utiliza la sentencia IF - THEN - ELSE, la cual nos permite escoger una de entre dos opciones, colocada de la siguiente forma:

```
IF   Expresión Booleana   THEN
                                Sentencia 1
                                ELSE
                                Sentencia 2;
```


Si (IF) se cumple la expresión Booleana entonces (THEN) se realiza la sentencia 1, sino (ELSE) se realiza la sentencia 2. Se ha de hacer notar que la sentencia IF elige una de las dos opciones y que en ningún caso se van a realizar las sentencias posibles.

Como dato sintáctico (Apéndice I), debemos destacar que después de la Sentencia 1 (antes de ELSE) no se coloca nunca el punto y coma (;), pues éste sólo se utiliza para separar dos sentencias y IF - THEN - ELSE constituyen una sola sentencia. Si pusiéramos un punto y coma al final de la Sentencia 1, el compilador estaría a la espera de una nueva sentencia, mientras que lo que recibiría sería un ELSE que por si mismo no constituye una sentencia, sino una parte de la sentencia IF.

El resultado de esta sentencia es Booleano (True o False), por lo cual a estas expresiones las llamaremos Booleanas.

* Ejemplo:

```
IF NUM < MAXINT THEN
    NUM := NUM+1
ELSE
    WRITE ('fuera de rango');
```

Se puede dar el caso en el que queramos ejecutar una sentencia si se cumple una determinada condición y si no se cumple continuar con el programa. Para representarlo tenemos dos opciones: una es no colocar nada después del ELSE y la otra es no colocar el ELSE. Las dos son equivalentes.

* Ejemplo:

```
IF NUM < MAXINT THEN
    NUM := NUM +1
ELSE;
```

```
IF NUM < MAXINT THEN
    NUM := NUM +1;
```

2.5.2. Sentencia CASE.

Esta sentencia nos permite escoger una alternativa de acción de entre una lista de alternativas.

El proceso se realiza asociando el valor de una variable, que llamaremos selector del CASE, con una etiqueta colocada en cada una de las opciones a elegir. El diagrama sintáctico se muestra en el apéndice I.

* Ejemplo:

```
CASE n OF
  1 : A := A+1;
  2: B := B+1;
  3: C := C+1
END;
```

En este caso n será una variable del tipo INTEGER y según sea su valor (1, 2, 3) se ejecutará la sentencia correspondiente (A := A+1, B := B+1, C := C+1 respectivamente).

Las etiquetas no tienen porque ser siempre INTEGER. Pueden ser de tipo caracteres, palabras (STRING), etc., siempre y cuando la variable de selección sea del mismo tipo (excepto del tipo REAL).

También podemos utilizar sentencias compuestas dentro de cada opción, encerradas siempre entre un par BEGIN - END.

* Ejemplo:

```
CASE Letra OF
1: WRITE ('Primera');
2: BEGIN
    WRITE ('Segunda');
    X := X-1
    END;
'D','d' : WRITE ('Cuarta')
END;
```

Si el valor de la variable de selección no coincide con ninguna de las etiquetas, se pasará a ejecutar la siguiente sentencia siguiendo, así, su curso secuencial.

La sentencia CASE permite realizar otra opción (ejecutar una sentencia o conjunto de sentencias) en el caso de que el valor de la variable de selección tampoco coincida con ninguna

de las etiquetas. Para ello colocaremos ELSE después de la lista de etiquetas seguido de la(s) sentencia(s) que queramos realizar

* Ejemplo:

```
CASE n OF
  1 : A := A+1;
  2: B := B+1;
  3: C := C+1
ELSE D:=D+1
END;
```

2.6. Estructuras repetitivas.

Mediante este tipo de estructuras también conseguiremos variar el orden secuencial del programa, al igual que con las anteriores estructuras alternativas.

2.6.1. Sentencia WHILE.

La sentencia WHILE realizará una sentencia determinada mientras se cumpla una expresión Booleana.

Antes de ejecutar por primera vez la sentencia, comprueba si la expresión Booleana es falsa o verdadera. Si es verdadera realizará la sentencia hasta que dicha expresión no cambie su valor. Cuando ésto ocurra el programa seguirá su camino. Si desde un principio la expresión Booleana es falsa la sentencia no se llega a ejecutar.

Podemos sustituir, si es preciso, la sentencia simple por un conjunto de sentencias (sentencia compuesta) colocadas entre un par BEGIN-END.

* Ejemplo:

```
Valor := 100;
WHILE Valor > 10 DO
  BEGIN
    Y := Y * 3;
    Valor := Valor - 1
  END;
```

La diferencia con respecto a la sentencia IF es que ésta última tan solo puede realizar la sentencia resultado de la expresión Booleana una vez, mientras que en una sentencia WHILE,

la sentencia del bucle se podrá realizar más de una o ninguna vez. La IF se utiliza para "elegir" un camino del programa y la WHILE para "repetirlo".

Cada vez que se ejecuta el bucle diremos que hay una iteración del bucle.

La condición que hace que el sistema salga del bucle se llama la condición de terminación.

Al resolver problemas nos encontraremos con dos tipos fundamentales de bucles: los bucles controlados por contador y los bucles controlados por suceso.

Bucles controlados por contador.

En este tipo de bucles la iteración se realiza un número conocido de veces. Los bucles controlados por contador utilizan una variable llamada de control del bucle. El proceso es el siguiente:

- Inicializar la variable.
- Comparar la variable.
- Incrementar la variable.

El ejemplo anterior constituye un bucle controlado por contador, donde "Valor" es la variable de control del bucle. Esta variable siempre se inicializará fuera del bucle. Una vez inicializada se compara con la expresión Booleana para realizar o no realizar el bucle en función del resultado. Cada vez que se ejecute el bucle, el valor de "Valor" tenderá al de la expresión Booleana hasta igualarla, con lo cual saldremos del bucle.

Un bucle en el cual la expresión Booleana nunca se hace False se denomina infinito.

Bucles controlados por suceso.

En estos bucles la condición de terminación depende de algún suceso intrínseco al bucle, que en principio desconocemos.

Hay varios tipos:

- Bucles controlados por centinela: Una utilización del bucle muy usual es la de leer o procesar grandes listas de datos. En cada iteración del bucle se lee un dato. Para saber cuando debemos frenar el proceso utilizaremos un centinela.

Mediante este dato conseguiremos salir del bucle de forma que el proceso seguirá, mientras el dato leído no coincida con el centinela. Hemos de tener en cuenta que por ser una salida del bucle nunca puede ser un posible dato de una lista de parámetros a leer o procesar. De esta forma al centinela le asignaremos un valor dentro de la variable que queramos leer.

* Ejemplo:

Si quisiéramos leer los días de la semana hasta domingo, elegiríamos el centinela igual a "domingo", de forma que cuando el bucle lea "domingo" se parará la iteración.

Para dar un valor a la/s variable/s que forma/n al centinela, leeremos antes de empezar el bucle el primer conjunto de datos (lectura adelantada), de forma que al inicio del bucle se procese la información obtenida previamente, para al final del bucle hacer una nueva lectura de las variables.

Un problema que nos puede suceder es que olvidemos introducir el valor del centinela en la lista de parámetros a procesar. Si los datos son entrados por teclado se puede añadir el centinela por el mismo, pero si los datos son tomados de un fichero no va a ser posible, con lo cual una vez el bucle haya leído todos los datos del fichero, ante la nueva petición de dato por parte del bucle se produciría un mensaje de error.

Cuando un sistema que toma los datos de un fichero ha finalizado la lectura del último valor del mismo, se dice que se ha llegado al final del fichero (EOF).

- **Bucles EOF:** EOF es una función Booleana incorporada al PASCAL para determinar si se ha finalizado la lectura de un fichero. De esta forma EOF se hará True cuando el siguiente carácter a leer de un fichero es <eof>, carácter especial de final de fichero, que puede generarse desde teclado mediante CTRL-Z Su funcionamiento es similar al de <eoln> pero aplicado a ficheros.

Este tipo de ficheros se utiliza cuando se han de procesar datos de los cuales no conocemos su cantidad.

* Ejemplo:

```
WHILE NOT EOF DO
  BEGIN
    Readln (C1, C2, C3.....);
    .
    .
  END
```

Usaremos NOT EOF pues cuando EOF es True no hay más datos. Para utilizar Readln hemos de saber como están estructurados los datos en cada linea. Si usamos una sentencia Read nos pueden aparecer problemas.

* Ejemplo:

WHILE NOT EOF DO	<u>Datos</u>
BEGIN	
Read (C);	24 36 37<eoln>
.	<eof>
.	
END	
	POSICION MARCADOR

Vemos en el ejemplo anterior que al usar Read el marcador de posición (__) no llega al <eof> al leer el último dato, pues se sitúa en el <eoln>, con lo cual el bucle se ejecutará de nuevo produciéndose un error. O sea sólo se finalizará un fichero cuando el marcador esté en el carácter <eof>.

- Bucles EOLN: EOLN es una función Booleana al igual que EOF dada por PASCAL, para determinar si el último elemento de una lista de parámetros ha sido procesado.

Este tipo de función lo usaremos para controlar un bucle que procese caracteres.

* Ejemplo:

2.6.2.Sentencia REPEAT.

Actua igual que la sentencia WHILE sólo que en ésta la sentencia (simple o compuesta) siempre se ejecutará como mínimo una vez y se repetirá hasta que se cumpla una expresión Booleana. El diagrama sintáctico de esta expresión lo podemos ver en el Apéndice I.

* Ejemplo:

```
REPEAT
  A := A+1;
  B := B+1;
  X := A+B
UNTIL X > 10;
```

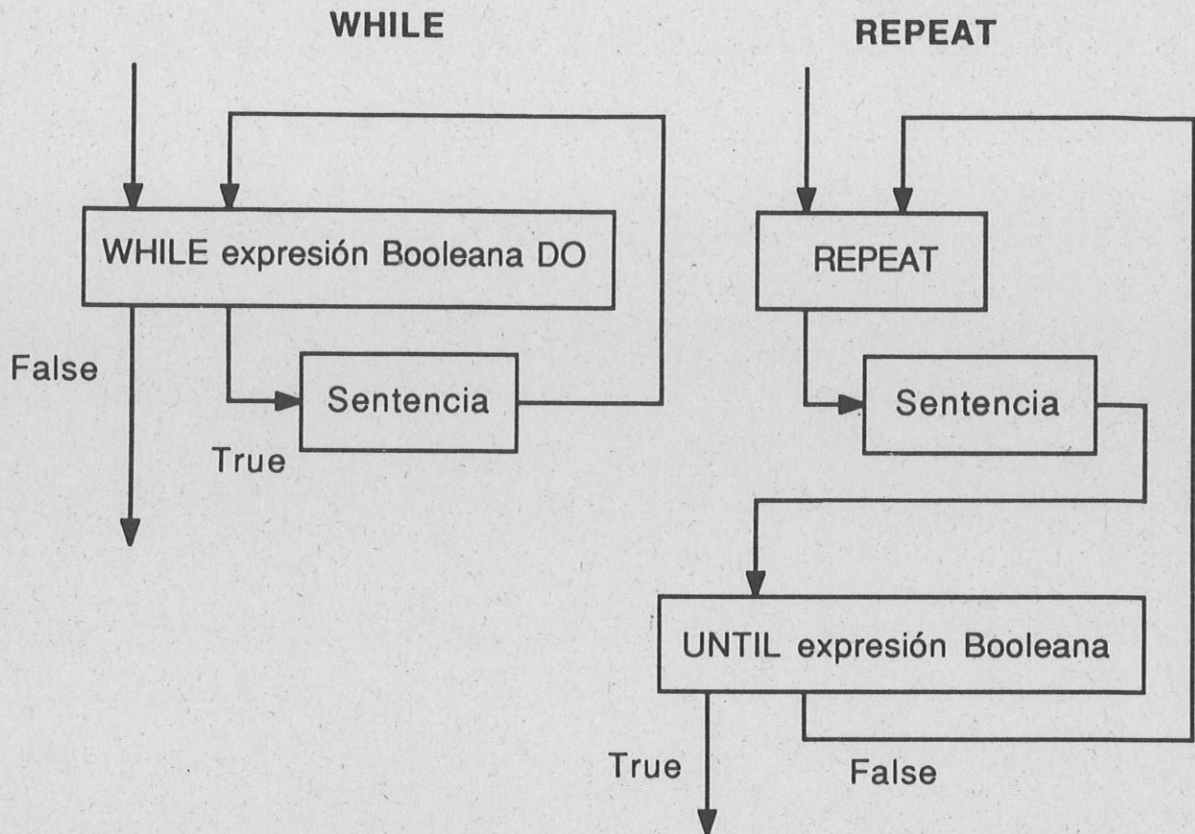
Debido a que UNTIL delimita la sentencia, no es necesario colocar el par BEGIN - END.

La diferencia con WHILE es que ésta última continua la iteración mientras la expresión Booleana sea True y REPEAT lo hace mientras sea False.

Diremos que la sentencia WHILE es un bucle *pretest* porque compara la condición del bucle antes de ejecutar el contenido del mismo. Análogamente una sentencia REPEAT diremos que es un bucle *posttest* pues hace lo contrario.

Se ha de hacer notar que la sentencia WHILE utiliza la expresión Booleana para decidir cuándo entramos en el bucle mientras que la sentencia REPEAT lo hace para decidir cuándo salir del mismo.

Para ver la diferencia entre ambas estructuras representaremos sus respectivos diagramas:



2.6.3. Sentencia FOR.

Su misión es la de repetir sentencia determinada hasta que la variable de control (llamada en este caso contador) se iguale a un valor dado. Al contador se le asigna un valor inicial y el valor final al que tiene que llegar puede venir determinado por una variable. Su diagrama sintáctico se puede ver en el Apéndice I.

* Ejemplo:

```
FOR Contador := 1 TO N DO
    Sentencia;
```

El contador se incrementa en 1 automáticamente hasta llegar a N.

Con esta instrucción también se pueden utilizar variables compuestas colocadas entre un par BEGIN - END.

Esta sentencia presenta la posibilidad de que el contador en vez de incrementar en 1 su valor se decremente en 1. Para ello sustituiremos la palabra TO por DOWNTO.

Así:

```
FOR Contador := MAX DOWNTO 1 DO
    Sentencia;
```


es equivalente a

```
FOR Contador := 1 TO MAX DO  
  Sentencia;
```

2.6.4. Criterios para la elección de una estructura repetitiva.

Comentaremos algunos criterios básicos para la elección de una de las tres sentencias repetitivas.

- Utilizaremos FOR cuando tratemos un bucle controlado por contador. Si el bucle estuviera controlado a su vez por un contador y por un suceso, o bien el orden de crecimiento o decrecimiento del contador fuera distinto de 1, usaríamos la sentencia WHILE o REPEAT.
- Utilizaremos REPEAT cuando el bucle esté controlado por un suceso y éste se ejecute al menos una vez. Si se desconoce la primera ejecución usaremos WHILE.
- Si ante un problema podemos usar tanto WHILE como REPEAT, utilizaremos uno u otro en función de como debemos actuar sobre la iteración. Si consideramos cuando continuar la iteración usaremos WHILE. Sin embargo, si lo que consideramos es cuando parar una iteración usaremos REPEAT.
- Ante una duda elijiremos WHILE.

2.7. Subprogramas.

Hasta ahora en PASCAL, hemos visto las tres estructuras básicas secuencial (secuencia), condicional (selección) y repetitiva (bucle). La estructura restante es la de un subprograma, que es simplemente un programa que está disponible para ser usado dentro de otros programas.

De esta forma, un programa realizará una llamada a un subprograma para realizar las instrucciones que contenga y una vez finalizado volverá al programa principal.

En PASCAL un subprograma puede ser de dos tipos:

- Procedimiento.
- Función.

La principal diferencia entre ambos es la forma en que son llamados. Una llamada a un procedimiento es una sentencia incluida en un programa, mientras una llamada a una función es una parte de una expresión.

2.7.1.Procedimientos.

Un procedimiento es lo mismo que un programa con las siguientes diferencias:

- La cabecera PROGRAM se sustituye por PROCEDURE.
- La última sentencia END va seguida de punto y coma (;).

De esta forma una cabecera PROCEDURE da el nombre a un procedimiento y lista las variables (parámetros) que sirven como su entrada y salida.

* Ejemplo:

Una cabecera de un procedimiento sería algo como lo siguiente:

```
PROCEDURE Suma (VAR Número: Integer);  
BEGIN  
.  
.  
.  
END;
```

Entre los paréntesis de la cabecera de un procedimiento habrá una expresión, parecida a la declaración de variable de la cabecera de un programa. Esta expresión recibirá el nombre de declaración de parámetros formales. En el ejemplo anterior hay un parámetro formal, Número, del tipo Integer.

En un programa declararemos los procedimientos antes del programa principal y los llamaremos cuando su uso sea necesario. Para llamarlos lo único que deberemos hacer será citar el nombre asociado al procedimiento junto con la variable que se desea enviar a ejecutar al procedimiento.

Su funcionamiento se podría comparar al de una sentencia Read, pues cuando un programa se encuentra una sentencia Read, lee o recibe un dato del exterior. Cuando un programa se encuentra con una sentencia que equivale a un procedimiento, lo que hace es enviar la variable al mismo para ejecutar el programa del procedimiento y volverla a introducir en el programa principal. O sea la diferencia entre un Read y un envío a procedimiento es que mientras que en la primera se recibe una variable del exterior que todavía no ha sido usada en el programa, cuando se vuelve de un procedimiento lo que se lee es una variable (parámetro formal) que ya ha sido entrada al programa (asignación o lectura exterior) y que en el procedimiento se le ha realizado un determinado tratamiento.

Se podrán usar la cantidad necesaria de procedimientos en un programa, siempre valorando el hecho de que la misión del mismo es la de clarificar el diseño de un programa (visualización del análisis descendente). O sea usaremos procedimientos cuando éstos nos ayuden a entender el programa global. Además también los usaremos cuando queramos realizar

Automáticamente el primer parámetro actual se asignará al primer parámetro formal de la cabecera del procedimiento. Una vez éste haya sido tratado se devolverá al programa reasignándole al valor actual el último valor del parámetro formal.

De cierta forma el parámetro formal "Número" es una manera de reservar un espacio en la declaración del procedimiento al parámetro actual "Letra".

Como la asignación entre parámetros formales y actuales es uno a uno tantas veces como parámetros haya, significa que podremos variar siempre que queramos el nombre de los parámetros actuales.

El hecho de poder variar constantemente el nombre del parámetro actual, nos conlleva a decir que podemos utilizar un procedimiento desde cualquier punto del programa principal, llamándolo con cualquier variable que utilizaremos como parámetro formal y se asignará al parámetro actual del procedimiento.

Para resumir este bloque:

- diremos que para cada entrada de datos (parámetros formales) aún distinta, se activa el mismo proceso (procedimiento).
- el emparejamiento entre parámetros formales y actuales se hace por su posición en las respectivas listas de parámetros, o lo que es lo mismo, asignar cada parámetro actual a una posición de memoria, la cual asociaremos por posición en la lista de parámetros a la posición de memoria del parámetro formal a partir del cual obtendremos los datos necesarios para el desarrollo del procedimiento.

2.7.3.Procedimientos: declaración e invocación.

Un procedimiento está formado por una cabecera y un bloque. En la cabecera introducimos los parámetros formales que son las variables que nos permitirán la comunicación con el programa principal. Las variables que necesite el procedimiento para su uso interno las deberemos definir en una sección de declaraciones al igual que se hace para el programa principal.

El diagrama sintáctico de la construcción de un procedimiento se puede ver en el Apéndice I.

En un programa global los procedimientos se declaran después de las declaraciones de variables y antes del programa principal. Además si hay procedimientos anidados, todo procedimiento debería ser declarado antes de ser utilizado por otro subprograma.

Para llamar o invocar a un procedimiento desde el interior de un programa principal usaremos el nombre del mismo más las variables que se van a utilizar entre paréntesis, como una sentencia más en el programa.

O sea con una llamada a procedimiento transferimos el control del programa al mismo, ejecutándose su bloque de sentencias. Éste finalizará cuando se encuentre la última sentencia END, devolviendo el control al programa principal justo en la sentencia inmediatamente inferior a la llamada a procedimiento.

A la hora de escoger un nombre para un procedimiento es útil coger uno que indique algo sobre el funcionamiento del mismo.

A los parámetros formales vistos hasta ahora los llamaremos "parámetros por variable" o VAR, debido a su semejanza con una declaración VAR de un programa principal. Hay un segundo tipo de parámetros, "parámetros por valor", por el cual el procedimiento sólo recibe una copia del valor del parámetro actual y no la dirección de memoria como en el parámetro anterior, con lo cual el procedimiento no podrá variar el valor del parámetro actual.

Hemos de tener en cuenta que cuando un procedimiento ha finalizado la unión entre las listas de parámetros formales y actuales se corta, y se devuelve el control al programa principal.

Obviamente el tipo de dato definido para un parámetro formal ha de ser el mismo que se defina para su parámetro actual asociado. Además el número de parámetros en cada lista ha de ser el mismo. Si los parámetros emparejados no fueran del mismo tipo se produciría un mensaje de error.

Diremos que una variable se llama *variable local*, cuando sólo es accesible desde dentro del bloque en que se declaran. Así por ejemplo, si nosotros definimos una variable en un procedimiento que es un resultado intermedio en el proceso del mismo, esta variable no podrá ser escrita desde el programa principal, sino que sólo podrá ser vista desde el mismo procedimiento. En el caso de que la quisiéramos imprimir desde el programa principal se produciría un mensaje de error.

El valor que adquiere una variable local se mantiene sólo durante el tiempo en que el control pasa al procedimiento. O sea cada llamada a procedimiento es independiente de la anterior, con lo cual antes de empezarse éste, deberemos inicializar estas variables en su interior.

Si en un procedimiento calculamos un valor que ha de usarse en otra llamada al mismo, este valor lo deberemos guardar en el programa principal a través de un parámetro, puesto que las variables locales de un procedimiento no se guardan.

Diremos que una variable es *global* cuando se define en el programa principal y por ello es accesible desde todos los puntos del mismo.

Ahora analizaremos la diferencia entre los parámetros VAR y los parámetros por valor. En los primeros podría suceder que cambiáramos un parámetro actual sin desearlo, pues al transmitir la posición de memoria al parámetro formal cualquier dato que a éste se adjudicase directamente se asignaría al parámetro actual. Los parámetros por valor evitan este problema

pues lo que se transmite al parámetro formal es el valor del parámetro actual, con lo cual el valor de éste no cambiaría debido al procedimiento. Si el procedimiento no conoce la posición de memoria en que se halla un parámetro formal sino que lo único que sabe es su valor, éste no se puede cambiar.

Hasta ahora hemos definido tres tipos de parámetros: actuales, formales por valor y formales VAR.

Cuando a un procedimiento le enviamos un parámetro por valor, lo que hacemos es enviar una copia del valor del parámetro. Si el procedimiento desea cambiar este valor, modifica sólo el valor de la copia, no el del original. De esta forma cuando finaliza su ejecución los parámetros formales por valor son borrados, como si fueran variables locales. La diferencia con éstas es que sus valores iniciales no están indefinidos, sino que se definen como los parámetros actuales.

Ahora bien, si dado un procedimiento con parámetros formales por valor, cuando éste finaliza tanto las variables locales como los parámetros por valor se borran y no pueden utilizarse para devolver información al programa principal ¿para qué sirven este tipo de parámetros?. Estos parámetros se introducirán cuando lo único que queramos hacer con un valor es introducirlo como dato al programa. Por ejemplo, si a un procedimiento le entramos tres datos procedentes del programa principal, dos de los cuales no deberemos modificar y otro que si, haremos lo siguiente: declararemos como parámetros formales por valor los que no queramos modificar y el dato a cambiar lo entraremos como parámetro formal VAR.

La razón de usar parámetros por valor y no siempre parámetros VAR es que nos eliminan los efectos laterales del programa, clarificando el diseño del programa y además sus parámetros actuales pueden ser expresiones.

Diremos que un efecto lateral sucede cuando en algún lugar del procedimiento se asocia a la posición de memoria de un parámetro VAR el valor de un resultado obtenido en el mismo, cambiando involuntariamente el valor del parámetro. Los efectos laterales serán evitados mediante el uso de parámetros por valor.

El uso de estos parámetros nos ayudan en la lectura del programa, puesto que nos permite saber qué valores vamos a poder modificar en el procedimiento y cuáles no.

Puesto que los parámetros VAR son posiciones de memoria, su parámetro actual asociado sólo podrá ser una variable. Sin embargo, como los parámetros por valor pasan el valor del parámetro, cualquier valor que tenga un parámetro actual podrá ser transmitido al procedimiento asociado a un parámetro formal, con lo cual los parámetros actuales podrán ser variables, constantes y expresiones.

La distinción entre los parámetros por valor y los VAR a la hora de escribirlos en una cabecera de procedimiento, es que los primeros no llevan la palabra reservada VAR delante del identificador. Su diagrama sintáctico puede verse en el Apéndice I.

* Ejemplo:

```
PROCEDURE Suma (VAR N1:Integer;
                N2:Real;
                VAR N3:Real;
                N4,N5:Integer;
                N6:Char);
```

En este ejemplo N1 y N3 son parámetros formales VAR y N2, N4, N5 y N6 son parámetros formales por valor.

* Ejemplo:

En el ejemplo siguiente vemos como asignamos dos valores a una misma variable. La variable B recibe en el programa principal un carácter "H" y en el procedimiento un carácter "D". El programa dará prioridad a la asignación hecha en el procedimiento. O sea cuando haya una variable que esté definida tanto en el interior del procedimiento como fuera de el tendrá precedencia la primera en todas las referencias que se le hagan desde el procedimiento.

* Ejemplo:

```
PROGRAM Semana (Input, Output);
CONST
  A = 5;
VAR
  C : Integer;
  B : Char;
PROCEDURE Dia (C: Integer);
BEGIN
  B := 'D';
  WRITELN ('A = ',A : 1);
  WRITELN ('B = ',B);
  WRITELN ('C = ',C :1)
END;
BEGIN
  B := 'H';
  C := 4;
  Dia (C)
END.
```

2.7.4.Reglas de ámbito de los identificadores.

Diremos que el ámbito de acceso de un identificador son todos los puntos desde los cuales puede ser accedido un identificador global y las reglas que rigen este acceso son las reglas de ámbito.

Diremos que un identificador es no local con respecto a un determinado procedimiento si no ha sido declarado en su interior. De esta forma cuando un procedimiento accede a un identificador no local, diremos que hay un acceso no local.

Las reglas de ámbito son:

- El ámbito de un identificador incluye todas las sentencias que van a continuación de su definición, dentro del bloque que contiene la definición. Esto es extendible a bloques anidados, excepto lo especificado en la regla siguiente.
- El ámbito de un identificador no se extiende a cualquier bloque anidado que contenga un identificador definido localmente con el mismo nombre.
- El ámbito de un parámetro formal es el mismo que el de una variable local del mismo procedimiento.

En el bloque de la página siguiente podemos ver como una sentencia del procedimiento B1, accedería a cualquier identificador declarado en B1, K0 y el programa principal pues todos estos posibles identificadores se hallarían en un mismo bloque. Sin embargo, esta misma sentencia no podrá acceder a identificadores de B4, PA, ME, pues para hacerlo debería entrar a los bloques desde el exterior.

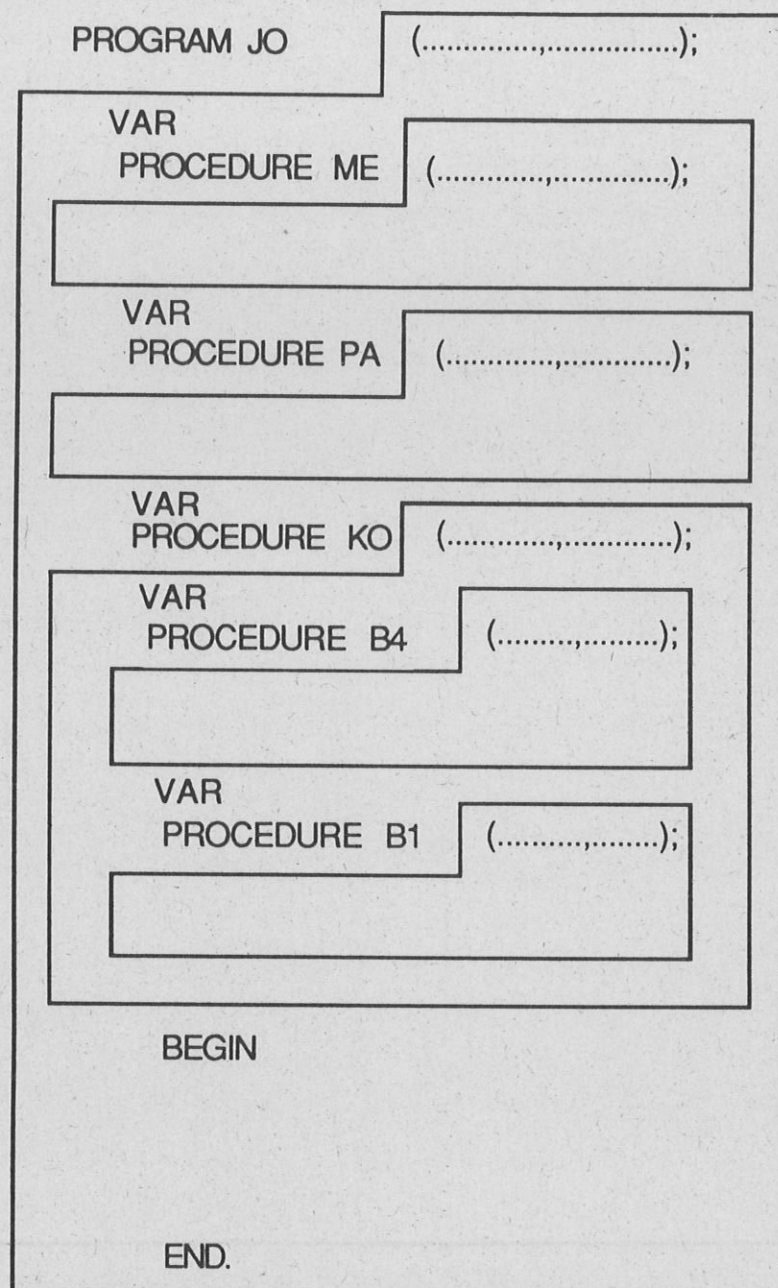
Si hubiéramos definido una variable con el mismo nombre en cada procedimiento, según la segunda regla el ámbito de esta variable estará restringido para cada bloque, o sea todos los procedimientos anidados redefinen localmente la variable.

2.7.5.Funciones: declaración e invocación.

En los apartados anteriores hemos visto el tipo procedimiento de subprograma. Ahora analizaremos otro tipo de subprograma llamado función.

La diferencia principal entre ambos subprogramas es cómo son llamados. Mientras que una llamada a procedimiento es una sentencia en un programa, una llamada a una función es parte de una expresión.

Las funciones se utilizan principalmente en situaciones en las que sólo se quiere devolver un valor como resultado, el cual se ha de usar directamente en una expresión.



*** Ejemplo:**

Supongamos que queremos dividir un número por la raíz cuadrada de otro número, desconociendo como se hace la operación raíz cuadrada. Ante esta situación deberíamos resolver esta operación analíticamente y a todo el proceso darle un nombre como SQRT, de manera que siempre que quisiéramos hallar la raíz cuadrada de un número, aplicaríamos directamente la función SQRT. Como quizás sabemos la función SQRT ya es una función predefinida en PASCAL.

Para definir una función:

- En la cabecera sustituimos la palabra PROCEDURE por FUNCTION.
- La lista de parámetros formales finaliza con dos puntos seguidos de un tipo de dato más un punto y coma.

Este último tipo de dato, llamado tipo de la función nos delimita el tipo resultado de la misma.

Otra diferencia con respecto al procedimiento es que cuando una función devuelve un valor no lo hace a través de un parámetro, sino a través del nombre de la función.

* Ejemplo:

Dividir un número determinado por el factorial de este número.

```

PROGRAM DIVISION    (Input, Output);
  VAR
    C: real;
    A: integer;
  FUNCTION FAC(N: integer): integer;
  VAR
    F: integer;
  BEGIN
    F:=1;
    while N > 1 do
      begin
        F:= F * N;
        N:= N - 1
      end;
    FAC:= F
  END;

  BEGIN
    writeln(' INTRODUCIR EL NUMERO ' );
    read (A);
    C:= FAC(A)/A;
    writeln;
    write(' EL RESULTADO DE LA OPERACION ES : ', C:1:2)
  END.
  
```


Vemos que a la función FAC le asignamos un valor el cual nos permitirá hallar un resultado que se devuelve al punto en donde se llamó la función. O sea la entrada de datos al subprograma función se hace directamente por asignación al nombre de la función. De esta forma el último valor que dentro del subprograma sea asignado al nombre de la función se devolverá al programa principal como resultado.

Cuando en un programa principal utilicemos el nombre de una función en una expresión se realizará una llamada a la función (también se podrá realizar desde dentro de la función).

El uso de una función no se restringe sólo a la devolución de un número, sino las podemos usar para devolver un resultado Booleano. Ésto será útil cuando la expresión de una sentencia alternativa sea muy compleja. Ante esta situación crearemos una función la cual llamaremos para que nos realice la expresión Booleana y nos devuelva el resultado True o False.

La función se genera en un programa para devolver un valor al programa principal. Cuando ésto no se cumpla y nos devuelva más de un valor al programa principal se produce un efecto lateral que debe evitarse. Cuando se deba devolver más de un valor utilizaremos un procedimiento. Una buena regla para evitar los efectos laterales en una función es la de no usar nunca parámetros formales VAR (excepto cuando el paso a una función es de un fichero), pues son los únicos que nos permiten cambiar el valor del parámetro.

Con las siguientes reglas podremos ayudarnos en la elección de un procedimiento o de una función a la hora de crear un subprograma:

- cuando haya más de una salida en un subprograma utilizar procedimientos.
- si sólo hay una salida y ésta es Booleana usar una función.
- si sólo hay una salida y el valor va a utilizarse inmediatamente usar una función.
- cuando haya duda utilizar procedimiento.
- si podemos usar ambos subprogramas elegir el más asequible a cada usuario en particular.

En el Apéndice II se muestra una tabla con las funciones estándar.

2.8. Tipos de datos.

Conocemos la existencia de cuatro tipos de datos, Integer, Real, Char y Boolean. Muchas veces con estos tipos de datos no podemos representar adecuadamente los valores de un programa. Gracias al lenguaje PASCAL podremos definir unos nuevos tipos.

Los datos que podemos incluir en los cuatro tipos anteriores reciben el nombre de escalares y presentan dos características comunes: son indivisibles y están ordenados.

Diremos que un tipo de dato es ordinal cuando cada valor, excepto el primero, tiene un único predecesor y un único sucesor, excepto el último.

A estos tipos de datos ordinales les podemos aplicar tres operaciones adicionales que en PASCAL están implícitas en funciones.

1.- Función ORD: su misión es la de dado un dato ordinal devolver el número entero que éste representa en la lista donde está ordenado. Si el parámetro es un número entero nos devolverá su mismo valor, pues será el orden que representa la ordenación. Si es Booleano será cero (false) o uno (true). Si el parámetro es char, el resultado será el lugar del mismo en la ordenación de caracteres (secuencia de cotejo) del código o conjunto de caracteres al que se refiere.

De esta forma con la función ORD podemos convertir un carácter en su equivalente numérico. Para un primer identificador listado el resultado de la ORD es un cero.

2.- Función PRED: su misión es la de dado un dato ordinal, hallar su inmediato predecesor. Se produciría un error si éste fuera el primero de la lista.

3.- Función SUCC: parecida a la anterior con la diferencia de que el valor devuelto es el inmediato sucesor.

4.- Función CHR: esta función, solo definible sobre valores tipo CHAR, toma una posición ordinal entera y devuelve el caracter correspondiente a esa posición.

2.8.1. Tipos ordinales. Normalizados, enumerados y subrango.

En la sección de declaraciones de un programa PASCAL, podemos incluir una sección TYPE antes de VAR, en la cual podremos definir nuevos tipos de datos. El diagrama sintáctico se muestra en apéndice I.

Los tipos de datos pueden ser:

1.- Normalizados: son los vistos hasta ahora.

2.- Enumerados: una forma de escribir los datos es referenciar al conjunto al que pertenecen. Otra forma de definir un conjunto de valores, es listar todos los valores que forman parte del mismo. Podremos definir nuevos tipos de datos ordinales, enumerando los valores literales que los forman.

* Ejemplo:

Tipo enumerado formado por nombre de ciudades.

TYPE

ciudades= (Barcelona, París, Londres, Milán);

VAR

grandes, pequeñas: ciudades;

Una sentencia de asignación del programa es al siguiente:

grandes:= Barcelona;

Hemos de diferenciar entre la sección de tipos y la de variables.

En la sección se variables introduciremos unos valores asociados a unos tipos de datos que a lo largo del programa podrán ser asignados a un determinado valor, el cual formará parte del grupo de datos al que los hayamos asignado.

En el ejemplo anterior hemos introducido un tipo de datos, ciudades, formado por una serie de elementos. Hemos definido dos variables, grandes y pequeñas, cuyos posibles valores pertenecerán a este tipo de datos. De esta forma a estos dos tipos de variables les podremos asignar cualquier valor del conjunto ciudades, tal y como hacemos en la sentencia de asignación.

La ordenación de los identificadores en la lista es por el mismo orden en que estan definidos en ella. Por ser ordinales podremos aplicar las tres funciones anteriores.

3.- Subrango: en PASCAL podemos definir un tipo de datos llamado subrango, que sea un rango especificado de un tipo de datos ordinal existente. Éste, llamado tipo residente o base, puede ser enumerado o normalizado.

Para definir el subrango (ver apendice I), mostraremos el valor inferior y el superior del subrango entre dos puntos seguidos.

* Ejemplo:	<u>incluye</u>	<u>tipo residente</u>
N=4..8	4,5,6,7,8	INTEGER
grandes=Barcelona ..Londres	Barcelona,Paris,Londres	CIUDADES

Observamos que no hay paréntesis en la definición de tipos de datos subrango.

La existencia de este tipo de datos ayuda en la detección de un valor que se ha salido del rango de una variable. Por ejemplo, si un contador no debe pasar de 10 se hace un subrango limitado a 10 de forma que si el contador coge un valor mayor se producirá un mensaje de error.

2.8.2. Tipos estructurados.

2.8.2.1.-Array.

Estructura de datos cuyos componentes son homogéneos y se seleccionan mediante índices (variables).

*** Ejemplo**

```
PROGRAM vector (input,output);
CONST
    N=5;
TYPE
    rango=1..N;
    fila=array[rango] of integer;
VAR
    B:fila;
    conta:integer;
BEGIN
    for conta:=1 to N do read(B[conta]);
    for conta:=N downto 1 do write(B[conta])
END.
```

Mediante el programa anterior asignamos los valores de una lista a un array, leyéndolo posteriormente en orden inverso:

Introducimos los números 1,3,5,7,9 . Éstos se almacenan en el vector B en las posiciones 1,2,3,4,5, respectivamente. A continuación leemos el contenido de las posiciones empezando por la 5 hasta la 1 obteniendo por pantalla el siguiente resultado: 9 7 5 3 1 .

El tipo ARRAY “ fila “ se define en la sección TYPE usando la palabra ARRAY seguida de “ rango “ que es un tipo subrango que nos dará la logitud del array, más OF INTEGER lo cual nos indica que el array estará formado por valores enteros.

La forma de acceder a cada componente del array es asignándole un índice que nos indicará la posición del elemento en el array. En el ejemplo anterior B es el array y Conta es el índice.

Los tipos de datos array serán definidos en la sección TYPE de un programa y las variables de este tipo se declaran en la sección VAR del mismo. Una vez hemos declarado una estructura de array con un nombre, podremos definir el número de variables deseado de este tipo en VAR.

El índice dará la cantidad de componentes que habrán en ese array y cómo se accederá a cada uno de ellos y será un valor fijo definido a la hora de definir un array o antes. De esta forma un array será tratado como un único parámetro.

Para acceder a un elemento determinado de un array, utilizaremos el identificador de la variable array seguido de una expresión que debe ser un valor entre corchetes.

Los siguientes bloques de sentencias definen al mismo vector:

TYPE

Fila= ARRAY [1..10] OF INTEGER;

TYPE

Rango=1..10;

Fila = ARRAY [Rango] OF INTEGER;

El valor del rango de un array deberá estar siempre definido al inicio de un programa.

Cuando queramos representar datos cuya representación nos lleve a una tabla de filas y columnas, utilizamos una estructura de datos llamada array bidimensional. Esta estructura es extrapolable a la de array multidimensional, la cual no puede ser visualizada. De esta forma para acceder a una componente de la tabla precisaremos dos coordenadas, la fila y la columna.

* Ejemplo:

Para acceder a la tabla representada por el juego de barcos, precisaríamos:

TYPE

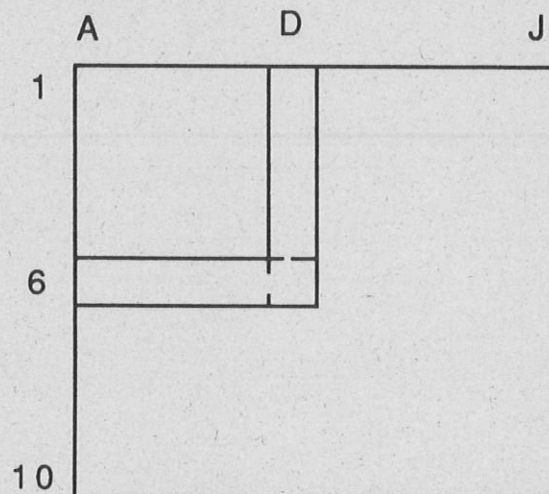
Fila = 1..10;

Columna = A..J;

Tabla = ARRAY [Fila,Columna] OF INTEGER;

VAR

Mapa : Tabla;



Para acceder a la posición 6D, escribiríamos Mapa [6,D].

Muchas veces se procede a inicializar una tabla, para lo cual accedemos a cada elemento individualmente para ponerlo a un valor, normalmente cero.

* Ejemplo:

Queremos realizar la suma de los elementos de cada fila de una matriz.

```
FOR Fila:=1 TO 10 DO
  BEGIN
    Total:=0;
    FOR Columna:=A TO J DO
      Total:= Total + Mapa [Fila,Columna];
    WRITELN ('Suma de filas=', Total)
  END;
```

En un problema específico, el número de dimensiones será el número necesario de características que nos definirán al dato.

Así si para acceder a un array bidimensional precisamos dos bucles, para hacerlo a uno n-dimensional precisaríamos n.

A la hora de diseñar un array hemos de tener en cuenta que todos sus componentes han de ser del mismo tipo de datos.

2.8.2.2.- Array empaquetado.

Una de las opciones que permiten los array es la de array empaquetado. La elección de esta opción se indica utilizando la palabra reservada PACKED seguida de la palabra que indica el tipo de dato estructurado, especificando la cantidad y el tipo de datos a utilizar.

* Ejemplo:

```
VAR
  Palabra : PACKED ARRAY [1..10] OF CHAR;
```

Esta opción de empaquetado es entendida por el compilador de la siguiente forma: colocará tantos caracteres como indica el contenido de los corchetes (de 1 a 10 caracteres en el ejemplo) en la posición de memoria asignada a la variable de este tipo (palabra). A este conjunto de caracteres empaquetados se les denomina cadena (string). Si los caracteres no

llegan a completar la longitud indicada en los corchetes, el compilador completa los espacios con blancos.

* Ejemplo:

```
BEGIN
  Palabra := 'COMPLETO';
```

en pantalla: 'COMPLETO__'

Si el dato introducido supera el número de caracteres del PACKED ARRAY el compilador coge sólo hasta el número indicado.

* Ejemplo:

```
Palabra := 'no está completo';
```

en pantalla: 'n o _ e s t á _ c o '

Tenemos también el tipo STRING cuyo funcionamiento es similar al de PACKED ARRAY. En éste sólo hay que indicar la longitud de la cadena.

* Ejemplo:

```
Palabra = STRING [10];
```

Cuando en un subprograma utilizamos un parámetro formal del tipo ARRAY lo debemos definir con su identificador de tipo y no de la siguiente manera:

```
TYPE
  matriz=ARRAY [1..3,1..3] OF INTEGER;
VAR
  MAT1: matriz;
PROCEDURE ASIGNA (MATRIX=ARRAY [1..3,1..3] OF INTEGER);
BEGIN
  _____
  NO

END;
BEGIN
  Asigna (MAT1);
END.
```

El compilador no admite la signación dando un mensaje de error.

2.8.2.4.- Registros.

Un registro es una estructura de datos cuyos componentes no son homogéneos y se seleccionan por sus nombres que serán fijos.

A cada dato le damos el nombre de *campo del registro* y puede ser de cualquiera de los tipos vistos en los apartados anteriores.

Vamos a ver cómo construir un registro. Si observamos el apéndice I vemos que para definirlos utilizamos la palabra reservada RECORD y un nombre o identificador que nos diferenciará un registro de otro. Después colocaremos todos los campos que contenga indicando a que tipo de dato pertenecen. La lista de campos quedará cerrada por un END.

* Ejemplo:

```
Nacimiento = RECORD
    DIA : 1..31;
    MES : 1..12;
    AÑO : INTEGER
END;
```

Observamos que cada campo va separado del otro por un punto y coma. En el último campo, antes del END, no se coloca nunca el (;).

Los registros se declaran en la sección TYPE tal y como indica su diagrama sintáctico (ver Apéndice I).

- Diferencia entre registro y array :

Vamos a ver cuales son las diferencias más significativas entre los dos tipos de datos:

- 1.- Un registro puede tener componentes de diferente tipo(cada campo puede ser de un tipo distinto) mientras que en un array sus componentes deben ser todos del mismo tipo.
- 2.- Los elementos de un registro se seleccionan por su nombre (identificador) mientras que los elementos de un array se seleccionan por subíndices que en general podrán ser variables o expresiones.

- Aplicación de los registros:

Los registros nos permiten gran flexibilidad a la hora de representar los datos y además nos permiten tratar su información como un 'todo' (el registro entero) o por partes (cada campo en particular). Si queremos trabajar con todo el registro sólo tenemos que utilizar el identificador de la variable registro. Para tratar la información de un campo debemos utilizar un

selector de campo, formado por el identificador de la variable registro seguido de un punto y el identificador de campo.

* Ejemplo:

```

TYPE
    Fecha=RECORD
        dia: 1..31;
        mes: 1..12;
        año: 1..2000
    end;
VAR
    data: Fecha;
BEGIN
    data.dia:=2;
    .
    .
    write (data.mes);
    .
    .
END.

```

La sentencia *write (data.mes)* nos escribirá en pantalla el contenido del campo *mes* perteneciente al registro *Fecha*, mientras que en la sentencia *data.dia := 2* se le asigna al campo *dia*, perteneciente al registro *Fecha*, el valor entero 2.

Si observamos la forma que tiene la declaración de un registro vemos que los campos se declaran igual que cualquier variable (identificador : tipo de dato). Diremos que, generalmente, todas las propiedades aplicables a una variable son aplicables a un campo de un registro: asignación, operaciones aritméticas, salida por pantalla, etc.

Como podemos ver en el ejemplo anterior, siempre es necesario declarar una variable (en nuestro caso *data*) que sea del tipo registro (*Fecha*) para realizar todas las operaciones en el bloque de sentencias. Nunca utilizaremos el identificador del tipo registro directamente como si de una variable se tratara.

- Registro de registros:

Según el apartado anterior, un campo de un registro puede ser definido de cualquiera de los tipos de datos que conocemos. Si definimos uno o más campos del registro, como tipo RECORD obtenemos un registro de registros:

* Ejemplo:

```

TYPE
    tiempo = RECORD
        hora : 0..24;
        minutos : 0..60;
        segundos : 0..60
    end;
    semana = RECORD
        time : tiempo;
        día : 1..7
    end;

```

El registro semana está compuesto por dos campos: el campo día que es del tipo subrango de enteros y el campo time que es del tipo tiempo siendo éste un registro compuesto por los campos hora, minutos y segundos.

Si tenemos la siguiente variable :

```
VAR
```

```
    estado : semana;
```

y queremos introducir un dato en el campo hora, lo haremos de la siguiente forma:

```
    estado.time.hora := 11
```

o queremos sacar por pantalla el valor del campo minutos y del campo segundos:

```
    write ('minutos:',estado.time.minutos,'segundos:',estado.time.segundos);
```

- Array de registros:

PASCAL nos permite 'combinar' tipos diferentes de datos como pueden ser arrays y registros.

Para ello definiremos un array cuyos componentes serán registros obteniendo, así, una lista (o una matriz) de registros numerados.

* Ejemplo:

```

TYPE
    ficha = RECORD
        apellidos,nombre: PACKED ARRAY[1..6] OF CHAR;
        DNI : PACKED ARRAY[1..10] OF CHAR;
        username: PACKED ARRAY[1..15] OF CHAR;
        password: PACKED ARRAY[1..15] OF CHAR
    end;
    control = ARRAY[1..600] OF FICHA;
VAR
    alumno : control;

```

De esta forma podemos tener 600 posiciones siendo cada una de ellas un registro del tipo ficha con sus campos correspondientes (apellido, nombre, DNI, username, password).

Si queremos acceder a un dato cualquiera lo haremos de la siguiente forma:

```

BEGIN
    write (alumno[7].nombre);

```

Como alumno es del tipo control (array) será necesario indicar entre corchetes el valor del índice seguido del campo deseado (perteneciente al registro ficha). Por pantalla obtendremos un conjunto de caracteres (packed array[1..6] of char) pertenecientes al campo nombre del registro ficha de la posición 7 del array control.

A partir de esta idea podemos construir estructuras de datos tan complejas como queramos.

* Ejemplo.-

Array de tres dimensiones siendo cada componente un registro de tres campos cada uno de ellos del tipo array de registros.

- Sentencia WITH :

Hemos visto que para acceder a un registro es necesario colocar la variable registro seguida del identificador de campo. Se puede dar el caso en que el registro contenga gran cantidad de campos siendo los selectores de campo muy largos y molestos. La sentencia WITH nos permite reducir los selectores de campo de forma que sólo tengamos que definir la variable registro una sola vez y luego utilizando los identificadores de cada campo podamos acceder a los componentes del registro:

* Ejemplo.-

TYPE

```
registro= RECORD
    uno: integer;
    dos: real;
    tres: integer;
    cuatro: real;
    cinco: real
```

end;

VAR

```
número: registro;
```

BEGIN

```
write(número.uno);
write(número.dos);
write(número.tres);
write(número.cuatro);
write(número.cinco)
```

END.

BEGIN

```
WITH número DO
```

```
begin
```

```
    write(unos);
    write(dos);
    write(tres);
    write(cuatro);
    write(cinco)
```

```
end
```

END.

- Registros variantes:

Hasta ahora todos los registros que hemos visto tienen una estructura rígida, es decir, hay un mismo número de campos con los mismos identificadores y los mismos tipos

Cuando realizamos un programa nos encontramos, en algunas ocasiones, con la necesidad de crear una estructura más flexible. Por ejemplo, si queremos crear una ficha de una persona indicando sus datos personales, características físicas, etc... algunos datos dependerán del sexo de la persona y por tanto no tendrán que ser declarados en el registro. Para ello utilizaremos los registros variantes de la siguiente forma:

TYPE

```

tipo = (varon,hembra);
persona = RECORD
    nom,ape1,ap2 : PACKED ARRAY [1..10] OF CHAR;
    Ecivil : (S,C,D,V,SE);
    CASE sexo : tipo OF
        varon : (emilitar: CHAR; barba : BOOLEAN);
        hembra : ( cintura,estatura : REAL)
    END;
END;

```

VAR

```
ficha : persona;
```

Utilizaremos la sentencia CASE-OF para seleccionar una de las opciones. El selector del CASE vendrá definido por un identificador indicando, a continuación, de que tipo es. Después definiremos las etiquetas y en cada una de ellas colocaremos los campos necesarios con sus tipos correspondientes. La estructura del registro será :

	nom	ap1	ap2	ecivil	sexo	emilitar	barba
FICHA (varon)							

	nom	ap1	ap2	ecivil	sexo	cintura	estatura
FICHA (hembra)							

Observamos que el registro ficha siempre tiene siete campos, dos de ellos variantes segun sea el valor del campo sexo. Si no tuviéramos la opción de utilizar registros variantes tendríamos que haber definido el registro ficha con nueve campos.

- Operaciones con registros:

Los campos de los registros pueden ser operados igual que una variable ordinaria siempre y cuando cumplan las mismas condiciones descritas para variable. Tambien podemos utilizar los registros como un todo , es decir, utilizar el registro completo. Una operación importante es la de asignación que pasa el contenido completo de un registro a otro. Para ello se debe cumplir una regla muy importante : las variables registro deben ser del mismo tipo.

* Ejemplo :

```

TYPE
    Fecha = RECORD
        dia: 1..31;
        mes: 1..12;
        año: 1..2000
    end;
VAR
    pasado,futuro: Fecha;
BEGIN
    (* suponemos que los campos tienen valores iniciales*)
    futuro.dia:= pasado.dia;
    futuro.mes:=pasado.mes;    <=>    futuro:=pasado;
    futuro.año:=pasado.año
END.
    
```

2.8.2.5.- Conjuntos:

Un conjunto es una colección de datos de un mismo tipo ordinal. Los valores que puede tomar una variable de tipo conjunto vienen especificados por la definición de este tipo.

Con este tipo de datos PASCAL permite utilizar la notación matemática de conjuntos: pertenencia, inclusión, intersección, conjunto vacío, etc.

Los conjuntos se definen en la sección TYPE utilizando las palabras reservadas SET OF seguidas del tipo.

* Ejemplo:

```

TYPE
    número = SET OF 0..50;
    letra = SET OF CHAR;
VAR
    num : número;
    let: letra;
    
```


o también :

```

TYPE
    número = 0..50;
    letra = ('a','e','i','o','u');
VAR
    num : SET OF número;
    let : SET OF letra;
    
```

Para asignar valores a una variable del tipo conjunto debemos colocarlos entre corchetes:

* Ejemplo:

```

num := [1..10];
num := [1,2,3,4];
num := [1,2,3..20];
    
```

- Operadores:

+ Unión entre conjuntos.

* Intersección entre conjuntos.

- Diferencia entre conjuntos.

El resultado de las operaciones, que utilizan estos operadores, serán del tipo conjunto.

Si tenemos dos conjuntos A y B:

A <> B Los conjuntos A y B son distintos, al menos
un valor difiere entre A y B.

A = B A y B son idénticos.

A <= B A es subconjunto de B.

A < B Hay al menos un valor de B que no está en A.

A > B Hay al menos un valor de A que no está en B.

A >= B B es subconjunto de A.

IN Pertenencia.

El resultado de estas operaciones es del tipo BOOLEAN.

* Ejemplo :

```

TYPE
    lista=0..30;
VAR
    número1,número2,número3: SET OF lista
BEGIN
    número1:= [0..20];
    número2:= [10..30];
    if 5 IN número1 then write ('pertenece');
    número3:= número1 * número2
END.

```

En el ejemplo podemos observar dos operaciones de asignación de elementos a los conjuntos número1 y número2. A continuación tenemos una condición que se cumple si el valor 5 pertenece (IN) al conjunto número1. Por último el conjunto número3 quedará asignado con la intersección del conjunto número1 y número2, es decir, número3 <-- [10..20].

2.9. FICHEROS.

Cuando empezamos un programa colocamos la cabecera PROGRAM... al final de la cual ponemos unos identificadores los cuales nos indican como se realizará la comunicación entre el programa y el exterior.

Cuando un programa finaliza da como resultado una serie de datos, que quizás desearemos utilizar en el transcurso del programa en estos casos guardamos los datos en unos conjuntos llamados ficheros. De esta forma los valores de los datos se leen desde los ficheros, de forma que si los datos se reciben desde el teclado de un terminal se dice que los datos provienen del fichero INPUT y OUTPUT para el caso de salida por pantalla.

Cuando un programa obtiene una serie de resultados éstos se guardan en un fichero el cual puede ser usado para comunicarse con otros programas o con el mismo programa. Este tipo de ficheros se llaman ficheros externos y tienen que declararse en la lista de parametros de la sección de declaraciones del programa.

Puede darse el caso que en cierta parte del programa precisemos más capacidad de memoria para almacenar una serie de datos (ida a procedimientos, etc.). Para hacer esto utilizaremos un fichero interno. Este fichero será grabado internamente al programa y no se salvará cuando éste haya finalizado.

Para acceder a un fichero previamente deberá ser declarado, para lo cual en PASCAL podemos definir un tipo de datos FILE OF..., el cual nos especificará que el identificador a él

asignado es un fichero y el tipo de sus componentes. La forma de ser declarado se muestra en el apéndice I.

Una característica de esta estructura de datos es que los componentes de un fichero son accedidos de uno en uno empezando por el primero, de forma que cada vez que se ha accedido a un componente se pasa al siguiente (acceso secuencial). Si quisiéramos volver a un componente ya leído deberíamos inicializar el fichero.

Cuando usemos ficheros éstos deberán ser preparados antes de realizar sobre ellos una operación de lectura o escritura. Esto se hace con unos procedimientos estándar.

| **REWRITE ()**: deja a un fichero preparado para la escritura eliminando todos los componentes anteriores, colocando el marcador de escritura al comienzo del fichero y poniéndolo en modo de escritura.

| **RESET ()**: prepara el fichero para la lectura poniendo el marcador de lectura en el primer dato del mismo y poniéndolo en modo de lectura.

Un fichero que esté en modo de escritura no podrá ser usado con sentencias de lectura, y en uno que esté en modo de lectura no se podrá acceder a escritura.

Podremos modificar el modo de un fichero en cualquier momento, usando RESET o REWRITE, teniendo en cuenta que ésta última borra el contenido del fichero.

No es necesario, obviamente, colocar la sentencia RESET o REWRITE cuando los ficheros a utilizar sean INPUT y OUTPUT.

Cuando los datos a introducir en un programa provengan de un fichero y no del teclado de un terminal pondremos el identificador del fichero del cual sacaremos los datos en la cabecera, pero en cambio si los resultados sólo los guardamos en otro fichero, deberemos poner en la cabecera el identificador del fichero y OUTPUT. El lector puede preguntarse que sucedería en caso de error en el programa, sin el fichero OUTPUT declarado en la cabecera.

2.9.1. Ficheros de textos.

Este tipo de ficheros son de gran interés puesto que son el módulo de comunicación entre usuario y ordenador. Están compuestos por caracteres y marcas de fin de línea. En PASCAL existe el tipo estándar TEXT, que se utiliza para declarar ficheros de texto. Las operaciones que podremos realizar sobre los ficheros de texto serán las siguientes:

RESET (F): Prepara el fichero F para su lectura desde el comienzo.

REWRITE (F): Prepara el fichero F para poder escribir sobre él desde el comienzo.

EOF (F): Es la función que será True si el siguiente carácter a leer es <eof>. Sino False. (Final de fichero)

EOLN (F): Será true si el siguiente caracter es <eoln>. (Final de línea)

READ (F,V): Lee los valores del fichero F y los guarda en la variable V y avanza la posición del marcador de lectura hasta pasar este componente.

READLN (F): Igual que el anterior pero saltando de línea.

REWRITE (F,V): Añade al fichero F el componente de cuyo valor es el de la expresión V, volviendo la posición de escritura al nuevo fin de fichero.

WRITELN (F,V): igual que el anterior, pero escribe <eoln> en F antes de volver.

Vemos que las operaciones que ya habíamos definido sobre fichero INPUT y OUTPUT pueden usarse en cualquier fichero de texto sin más que poner el nombre del fichero como primer parámetro.

Cuando usemos sentencias de escritura y lectura en ficheros de texto, obviamente la expresión dada V, deberá ser compatible con el tipo del componente del fichero F.

Si realizamos una sentencia READ (F,V) a un fichero que ya está finalizado se producirá un error. Para evitarlo se proporciona la función EOF, que advierte la condición de fin de fichero.

El modo secuencial de funcionamiento de un fichero puede plantearnos una duda. Si restringimos el fichero a componentes tipo caracter, todas las operaciones son compatibles con la definición de fichero. De esta forma las operaciones de entrada y salida sólo actuarán sobre un caracter y EOLN y EOF advertirán sobre la existencia o no de los caracteres <eoln> y <eof> respectivamente en el marcador de línea. Si suponemos que hay datos numéricos en el fichero las operaciones READ y WRITE no son compatibles con la definición del tipo fichero. Esto es porque los componentes del fichero son un único caracter, mientras que un READ o WRITE con parámetros numéricos pueden acceder a más de un caracter.

Sabemos que pueden aparecer problemas al utilizar la función READ junto con la función EOF al introducir datos numéricos. Esto es debido a que el caracter <eoln> siempre va a continuación del último dato de un fichero de texto. De esta forma EOF no será true hasta que no se haya leído el último valor y el bucle intentará leer un dato inexistente. Esto se puede evitar leyendo los datos como caracteres para luego ser convertidos.

Hasta ahora sólo hemos visto los ficheros de textos pero hay otros distintos. Así podemos hacer las siguientes sentencias:

TYPE

```
a= 1..10;
b=1..15;
ap=PACKED ARRAY [b] OF CHAR;
name=PACKED ARRAY [a] OF CHAR;
tarjeta=RECORD
    nombre: name;
    apellido: ap;
    número: INTEGER;
    username: PACKED ARRAY [1..5] OF CHAR;
end;
ficha= FILE OF tarjeta;
apodo= FILE OF name;
```

VAR

```
f1: text;
f2: ficha;
f3: apodo;
```

f1 es un fichero de texto, f2 es un fichero del tipo ficha y f3 es un fichero del tipo apodo.

Los ficheros que no son del tipo texto no contienen el caracter <eoln> con lo cual no están divididos en líneas. Esto nos permite observar que EOLN, READLN y WRITELN no tienen significado. Las sentencias RESET, REWRITE y EOF tienen el mismo significado que en los ficheros de textos, mientras que READ y WRITE leen o escriben únicamente un componente del fichero. El marcador de fin de fichero funciona igual tanto para los ficheros de texto como para los que no lo son.

2.10. ESTRUCTURAS DE DATOS DINAMICAS.

Hasta ahora hemos visto estructuras de datos estáticas, es decir, estructuras que deben ser definidas dimensionalmente en la compilación. Muchas veces nos encontramos en la necesidad de trabajar con una lista de datos sin saber que cantidad de éstos necesitaremos utilizar. Para ello creábamos, por ejemplo, una variable array con una dimensión muy superior a la cantidad de datos que se puede preveer. Pero esta previsión es muy difícil de ajustar con lo que se nos plantea una seria limitación en la estructura.

Para evitar esta limitación podemos utilizar otro tipo de estructura: la estructura de datos dinámica, que consiste en poder aumentar o disminuir la cantidad de sus componentes durante la ejecución del programa.

2.10.1. Punteros.

Las variables del tipo puntero son las que utilizamos para crear estructuras de datos dinámicas.

Puntero es un tipo de datos simple que consta de un número no limitado de valores indicando, cada uno, la posición de una variable de un tipo dado.

Este tipo de datos se declara en la sección TYPE utilizando el símbolo $\wedge$, de la siguiente forma:



* Ejemplo :

TYPE

caracteres = 'A'..'B';

numeros = 1..9;

señas = RECORD

nombre : PACKED ARRAY[1..20] OF CHAR;

domicilio : PACKED ARRAY[1..30] OF CHAR;

edad : INTEGER

end;

cadena = ^caracteres;


```

serie = ^numeros;
datos = ^señas;

VAR

  apuntacadena : cadena;
  apuntaserie : serie;
  apuntadatos : datos;

```

Según el diagrama sintáctico podemos ver que cadena, serie y datos son tipos puntero.

En el bloque de variables tenemos :

Apuntacadena que es una variable del tipo puntero que apuntará a una variable del tipo caracteres.

Apuntaserie es una variable del tipo puntero que apuntará a una variable del tipo numeros.

Apuntadatos es una variable del tipo puntero que apuntará a una variable del tipo señas.

Una vez han sido declaradas las variables, el compilador crea tres celdas vacías cuyo contenido, más adelante, será una posición de memoria:



A continuación crearemos tres variables del tipo caracteres, numeros y datos para que sean apuntadas por sus correspondientes variables puntero. Este proceso se realiza mediante el procedimiento NEW() de la siguiente forma:

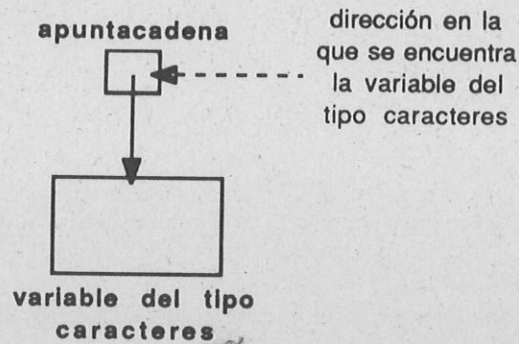
BEGIN

```

.
.
NEW (apuntacadena); ---> Crea una variable del tipo caracteres
.                          que será apuntada por apuntacadena,
.                          dejando su dirección en la celda co -
.                          rrespondiente.

```

END.



Este procedimiento incorporado se utilizará de la misma forma para las otras dos variables puntero.

A estas variables que se crean durante la ejecución del programa (paso por NEW) se las denomina variables dinámicas.

Vamos a ver como podemos acceder o referenciar a estas variables que hemos creado. Para ello utilizaremos la variable puntero de la siguiente forma:

`apuntacadena^ := 'B';` ---> A quien apunta la variable puntero `apuntacadena` asígnale el valor 'B'.

Obsevamos que `apuntacadena` apunta a una variable del tipo caracteres que es un subrango de CHAR, por tanto, la asignación del caracter 'B' es correcta.

De la misma forma podríamos escribir por pantalla el valor que contiene la variable apuntada :

`WRITE (apuntacadena^);` ---> Escribe en pantalla el valor contenido por la variable que es apuntada por `apuntacadena`.

2.10.2. Creación de una lista enlazada.

Una vez visto el procedimiento para crear variables dinámicas, vamos a ver de que forma podemos relacionarlas entre ellas para formar una lista de elementos.

Cada componente deberá estar relacionado con el siguiente mediante una variable puntero y además deberá contener la información que se le asigne. Por tanto, nos vemos en la

necesidad de utilizar registros de tal forma que uno de sus campos sea del tipo puntero y el otro (u otros) contenga la información deseada.

El proceso será el siguiente:

Definiremos en la sección TYPE los tipos de datos que vamos a utilizar:

TYPE

apuntaelemento = ^elemento;

elemento = RECORD

informacion: PACKED ARRAY [1..10] OF CHAR;

apuntador: apuntaelemento

end;

VAR

cl, auxiliar, apuntadato: apuntaelemento;

continuar: CHAR;

Tenemos dos tipos de datos: apuntaelemento y elemento:

Elemento es del tipo registro compuesto por dos campos : informacion y apuntador. El otro tipo es apuntaelemento que es del tipo puntero a elemento, es decir, las variables de este tipo apuntarán a registros. El campo apuntador es del tipo apuntaelemento (puntero) y nos permitirá relacionar un componente de la lista (registro) con su sucesor.

Vamos a ver como se generaría una lista de elementos a partir de los tipos de datos definidos anteriormente:

BEGIN

NEW (apuntadato); {1}

cl := apuntadato; {2}

repeat

auxiliar := apuntadato; {3}

NEW (apuntadato); {4}

auxiliar^.apuntador := apuntadato; {5}

read (continuar)

until continuar='N';

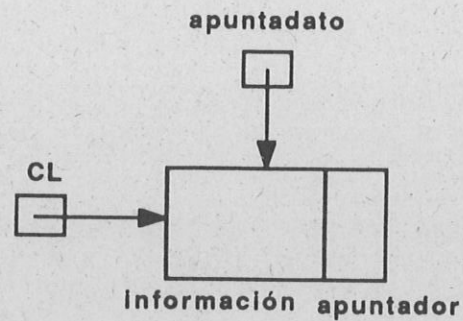
apuntadato^.apuntador := NIL; {6}

END.

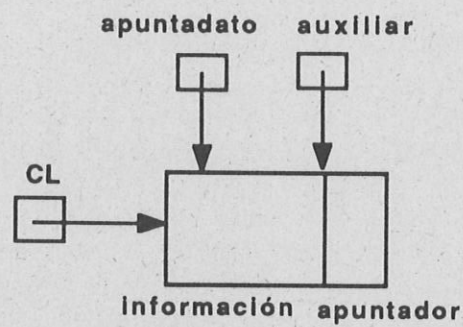
TIENE LA MISMA DIRECCION, PERO NO SE CREA NUEVA CASILLA

CREA UNA NUEVA CASILLA, Y LA DIRECCION DE ESTA NUEVA CASILLA QUEDA GUARDADA EN LA VARIABLE APUNTADATO

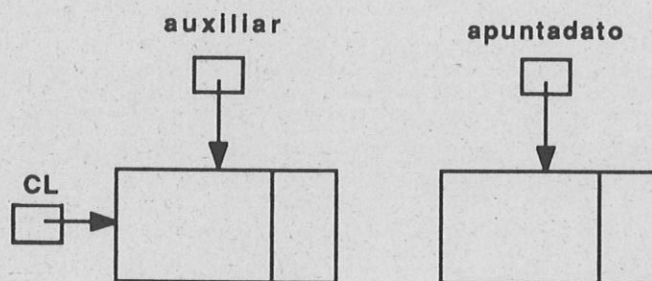
{1} y {2} Se crea la primera variable dinámica y se guarda la dirección en la variable cl (comienzo de lista) :



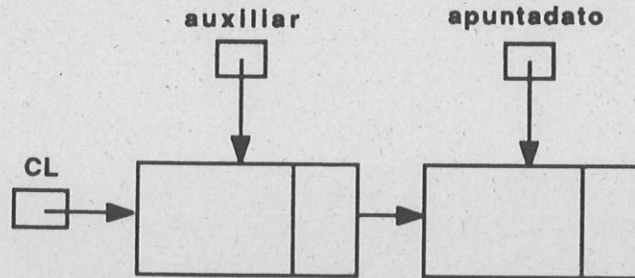
- {3} La variable puntero auxiliar apuntará a donde apunte apuntadato:



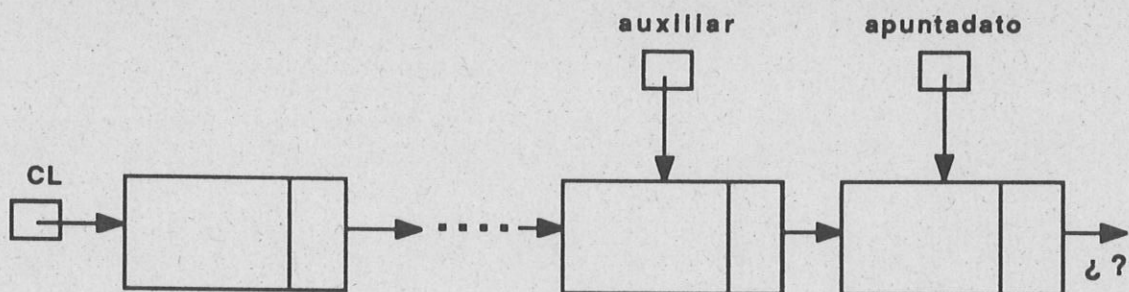
- {4} Se crea un nuevo componente :



- {5} La nueva componente es apuntada por el elemento anterior mediante el campo apuntador:

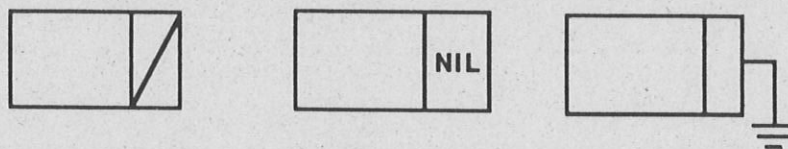


A continuación se pregunta si se quieren añadir más elementos a la lista. El bucle (repeat) se realizará hasta que la variable continuar sea 'N' creándose así, una lista enlazada:

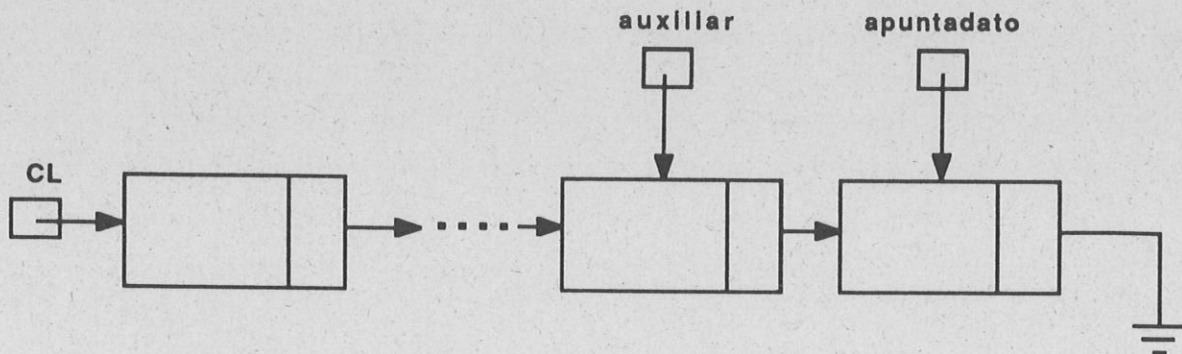


Una vez que se han creado todos los componentes de la lista y están debidamente enlazados es necesario indicar el final de la lista. Para ello debemos hacer que el último campo puntero apunte a 'nada'. Utilizaremos la palabra reservada NIL que es un valor que no apunta a ningún elemento. NIL pertenece a todo tipo de punteros y no es necesario declararla en ningún sitio.

NIL sólo será aplicable a variables de tipo puntero. Su representación en un diagrama de bloques, como el anterior, es :



nosotros utilizaremos esta última forma. Por tanto, la lista enlazada que hemos creado quedará :



Vamos a ver como introducimos los datos en el campo informacion :

```

apuntadato := cl;
while apuntadato <> NIL do
begin
    read (apuntadato^.informacion);
    apuntadato := apuntadato^.apuntador
end;

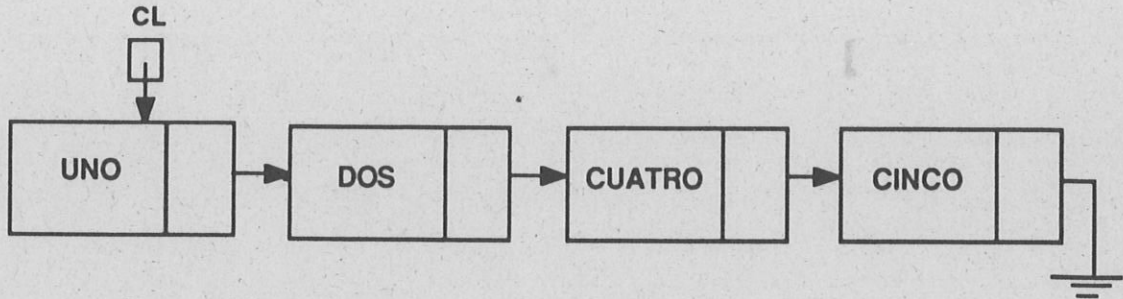
```

Apuntadato será la variable puntero que recorrerá toda la lista para hacer referencia al campo informacion y así, poder asignarle un valor. El primer paso será colocar apuntadato al principio de la lista. A continuación se irán recorriendo todos los componentes (mediante un bucle) dando valores al campo informacion (read) hasta llegar al final de la lista (NIL).

Este proceso se puede evitar si a medida que vamos creando las variables dinámicas vamos introduciendo los datos.

2.10.3. Inserción de componentes.

Si tenemos la siguiente lista:



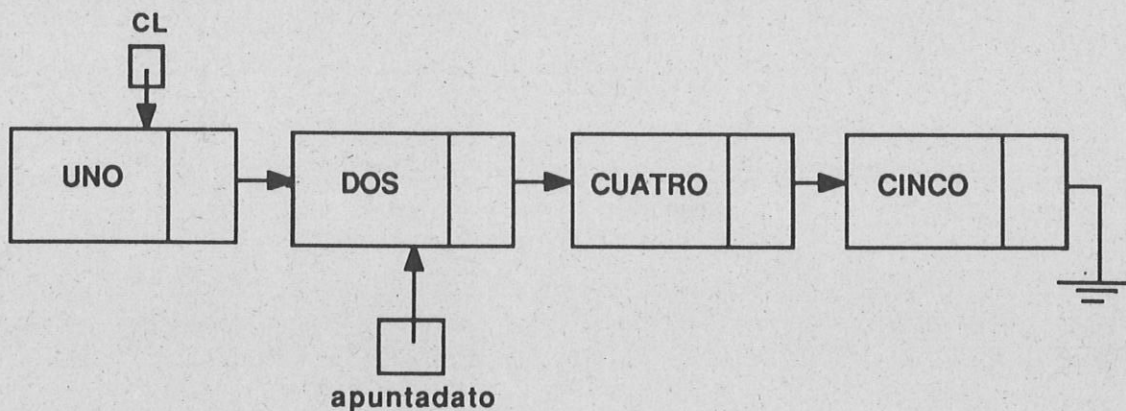
y queremos insertar un componente entre 'DOS' y 'CUATRO' el proceso será el siguiente:

```

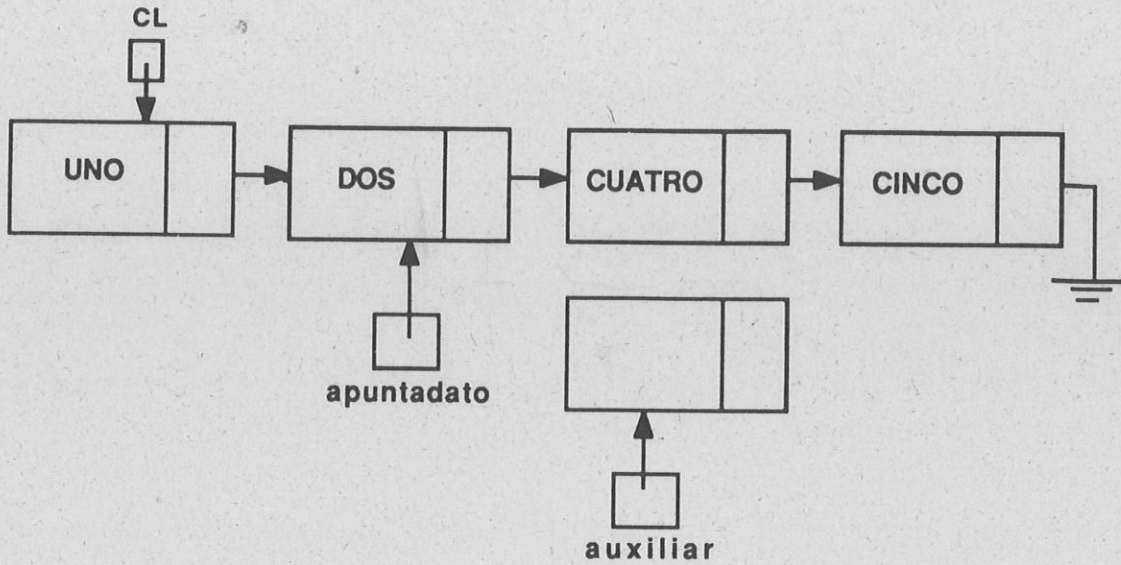
apuntadato:= cl;
while apuntadato^.informacion <> 'DOS' do
    apuntadato:=apuntadato^.apuntador;    {1}
NEW (auxiliar);                          {2}
auxiliar^.apuntador:=apuntadato^.apuntador; {3}
apuntadato^.apuntador:= auxiliar;         {4}
auxiliar^.informacion:= 'TRES';           {5}
  
```

Vamos a ver , gráficamente, la evolución del programa:

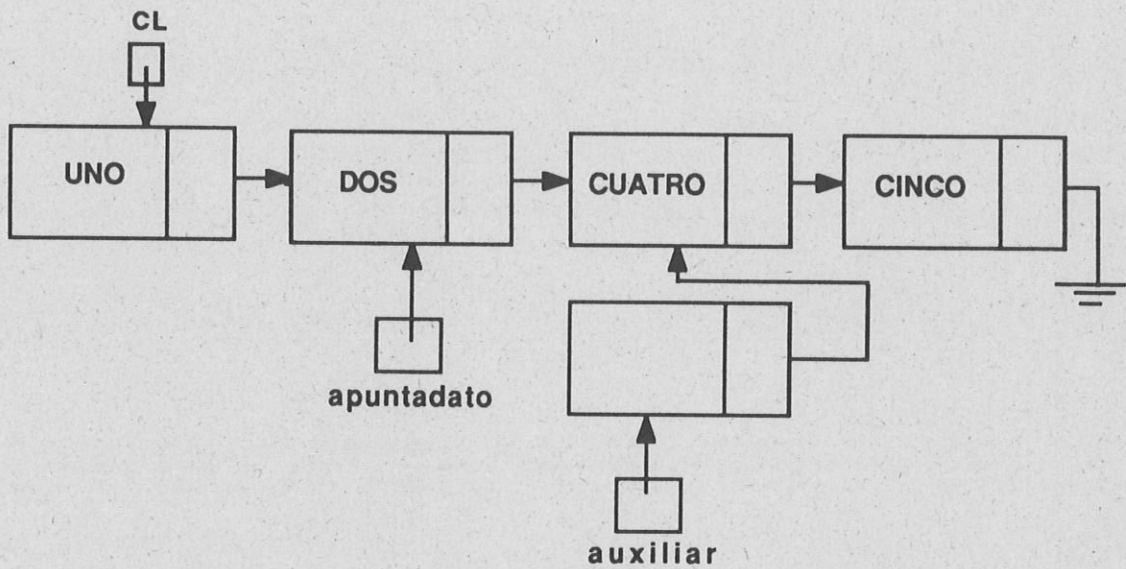
{1} Una vez colocada la variable puntero al principio de la lista se realiza una búsqueda del elemento deseado:

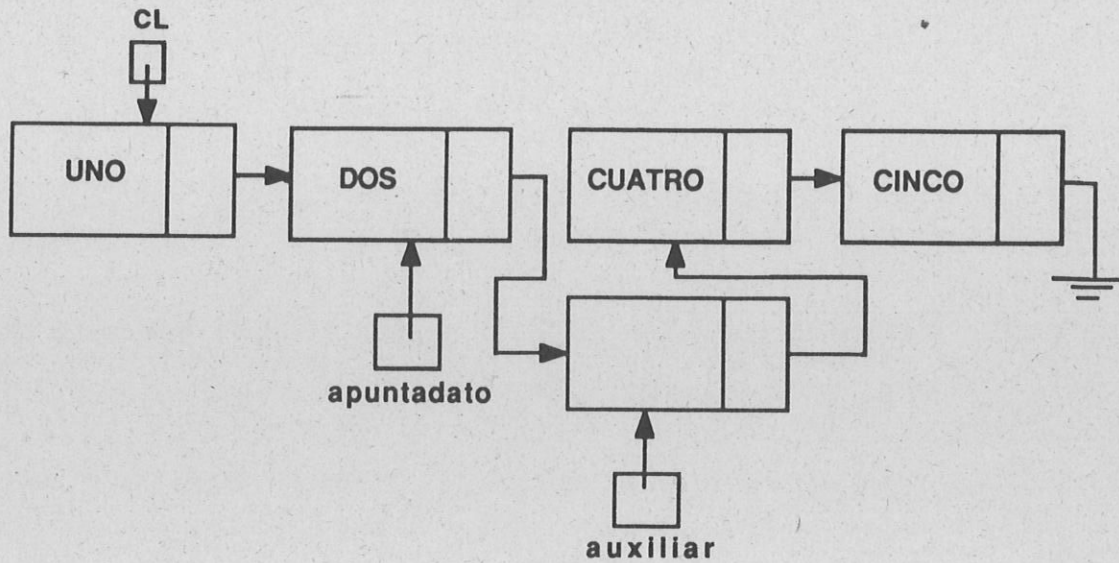


{2} Se crea la variable dinámica a insertar:

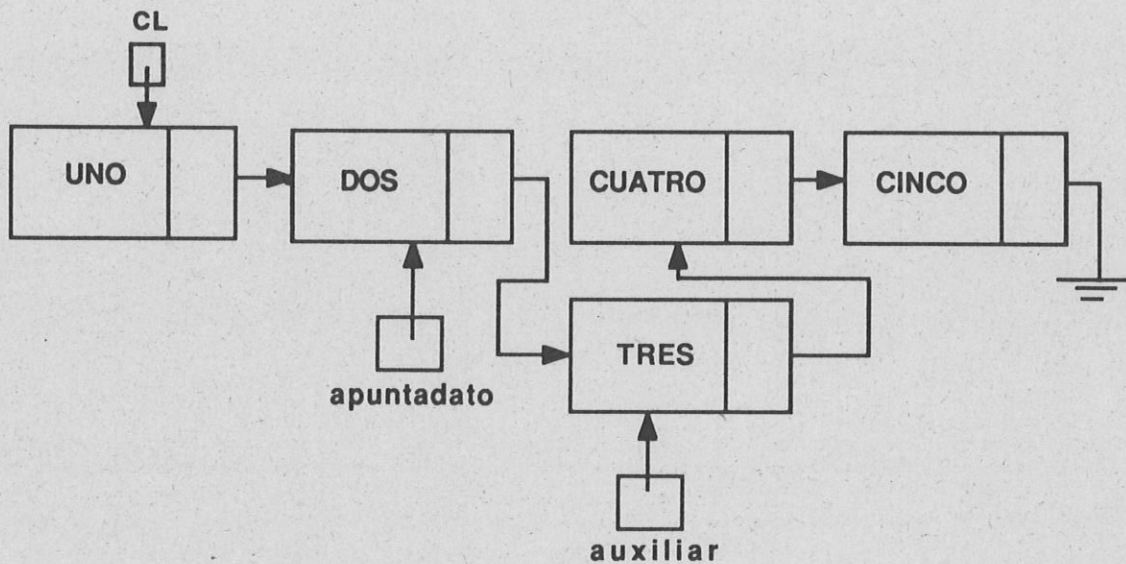


{3} y {4} se enlaza dicha variable en la lista:





{5} Se introduce el valor deseado en el campo informacion :



2.10.4. Destrucción de una variable dinámica.

Si partimos de la lista creada anteriormente y queremos suprimir el componente, por ejemplo, 'CUATRO' el proceso será el siguiente:

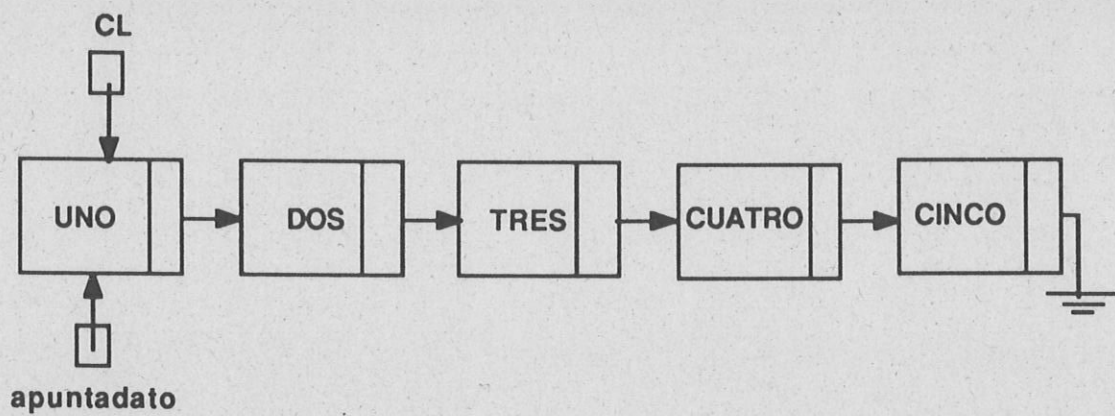
```

apuntadato:=cl;                                {1}
while apuntadato^.informacion <> 'TRES' do      {2}
    apuntadato:=apuntadato^.apuntador;
auxiliar:= apuntadato^.apuntador;              {3}
apuntadato^.apuntador:=auxiliar^.apuntador;    {4}

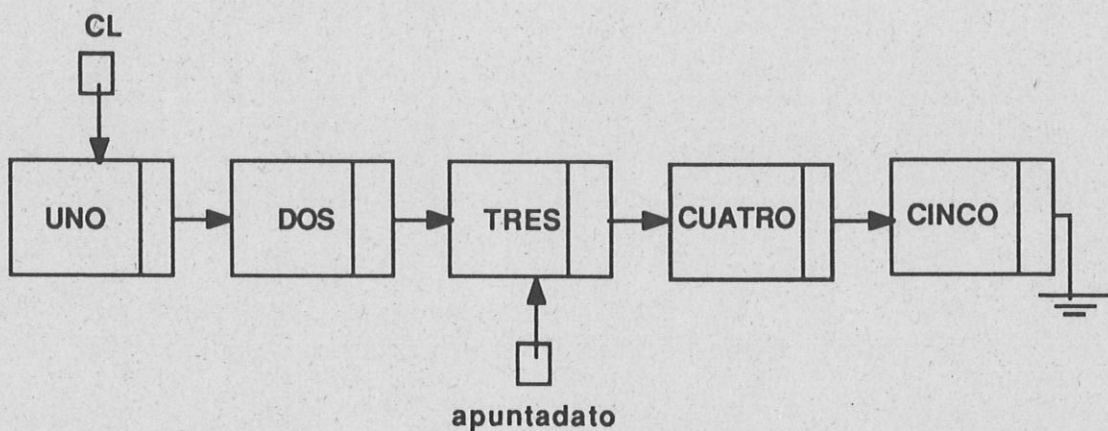
```

Vamos a ver este proceso gráficamente:

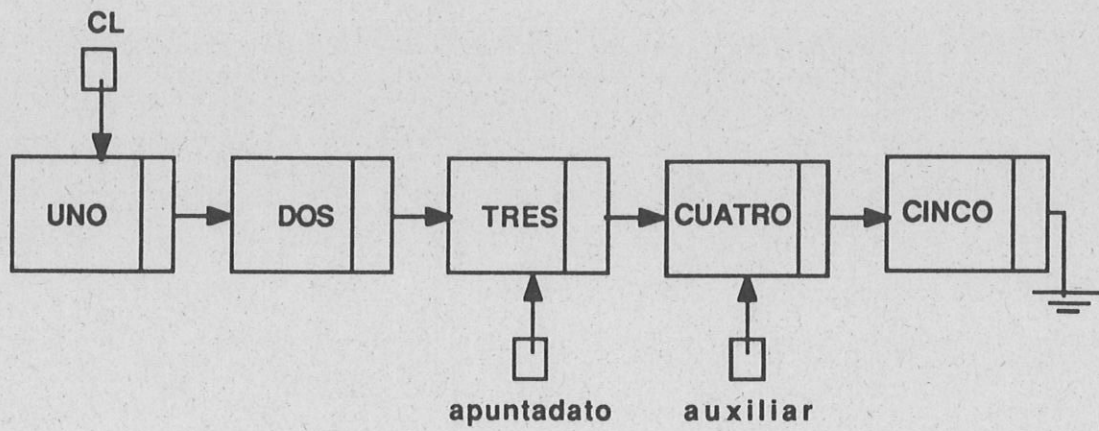
1) :



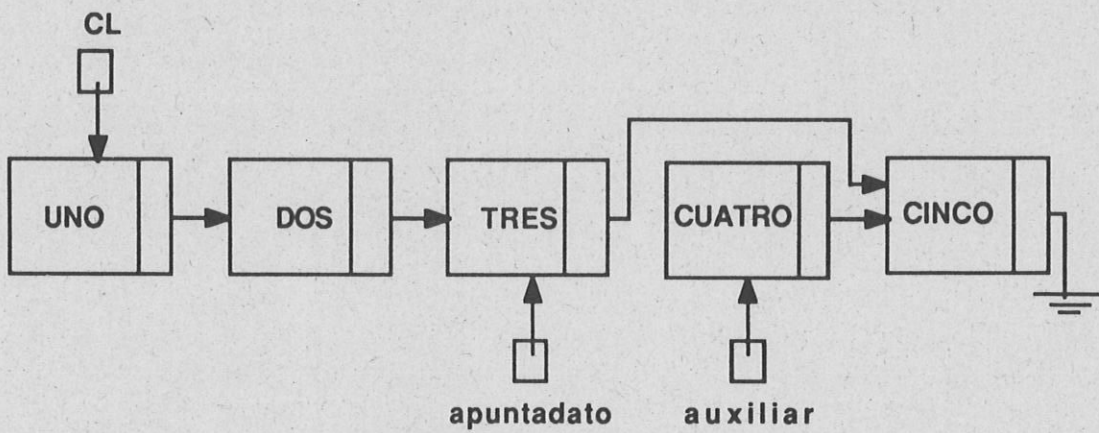
{2} :



{3}



{4} :

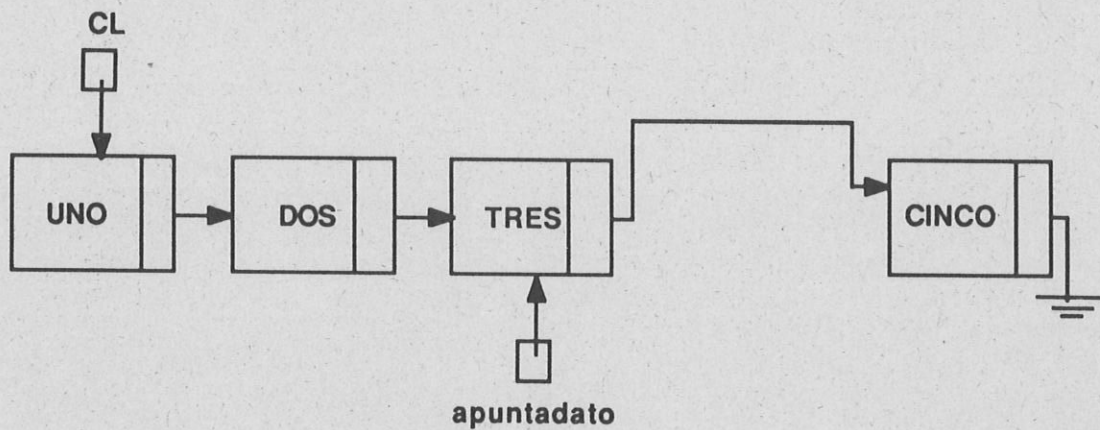


En este momento el componente CUATRO queda aislado de la lista pero sigue teniendo una dirección de memoria reservada y, por tanto, no puede ser utilizada para nada más.

Para evitarlo, PASCAL utiliza el procedimiento DISPOSE () :

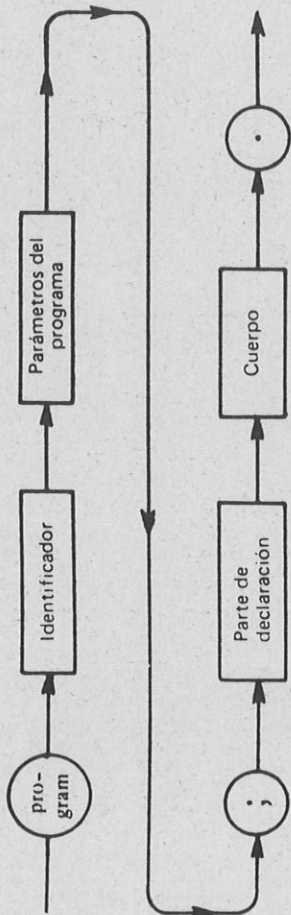
DISPOSE (auxiliar);

Devuelve la posición ,usada por la variable del tipo elemento, al sistema para que se utilice de nuevo.

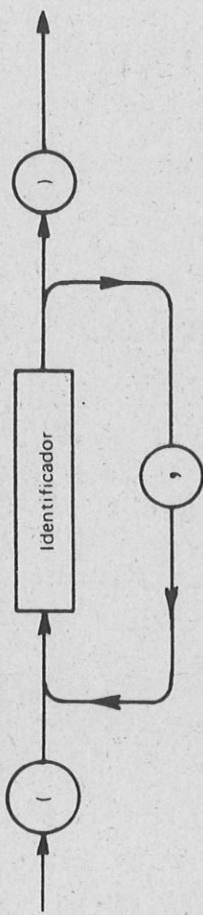


Diagramas sintácticos

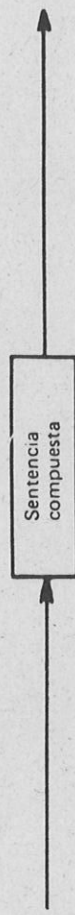
Programa



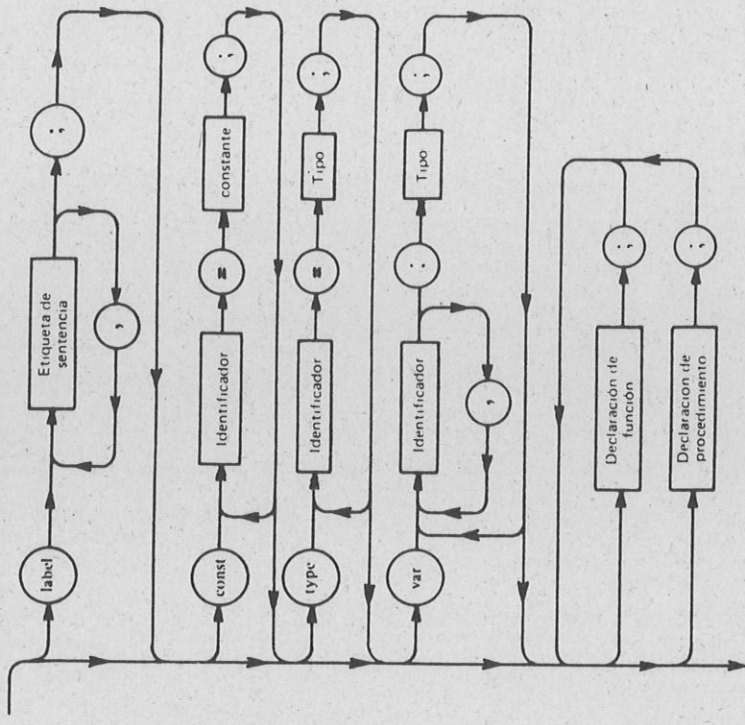
Parámetros del programa



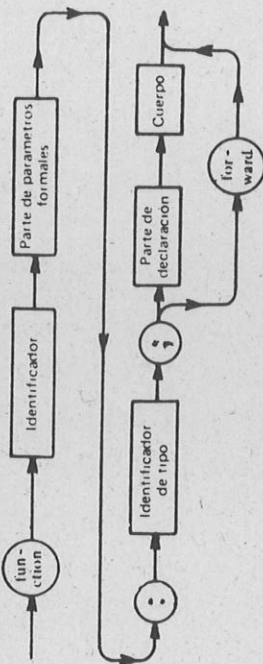
Cuerpo



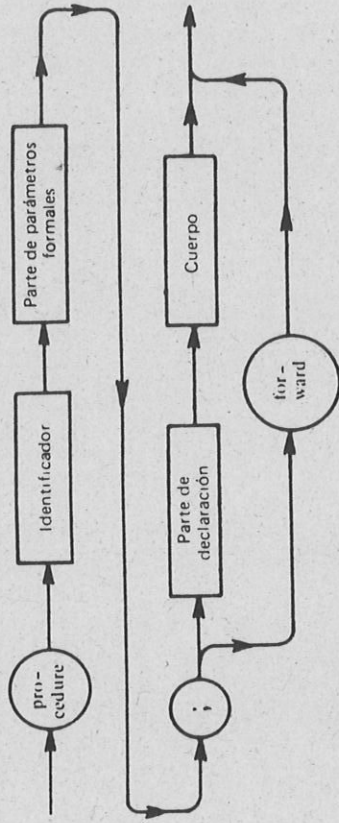
Parte de declaración



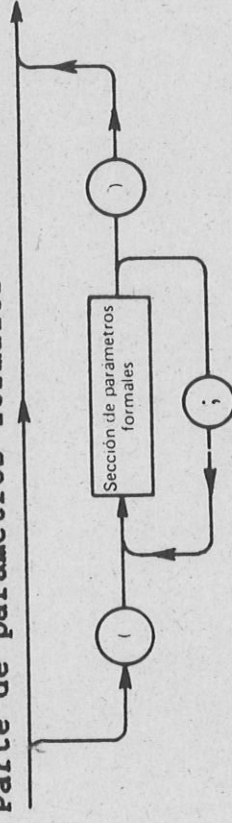
Declaración de función



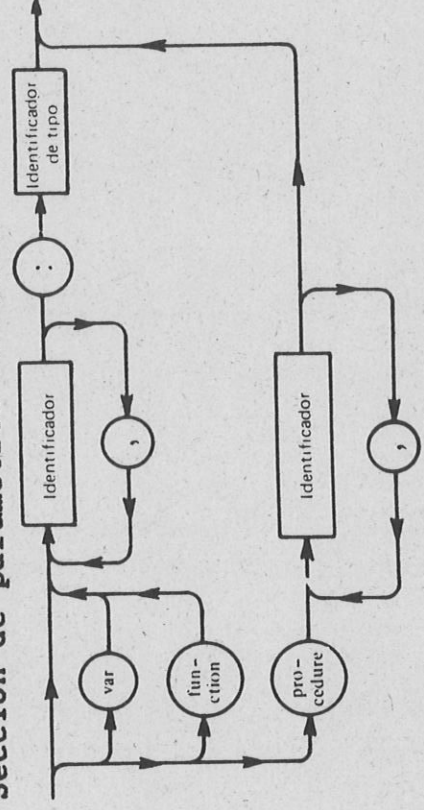
Declaración de procedimiento



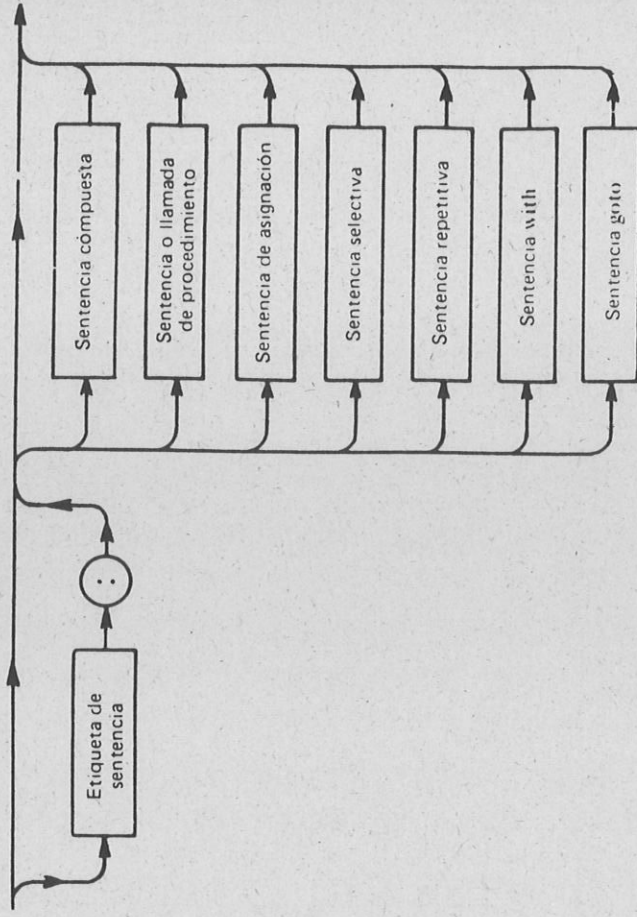
Parte de parámetros formales



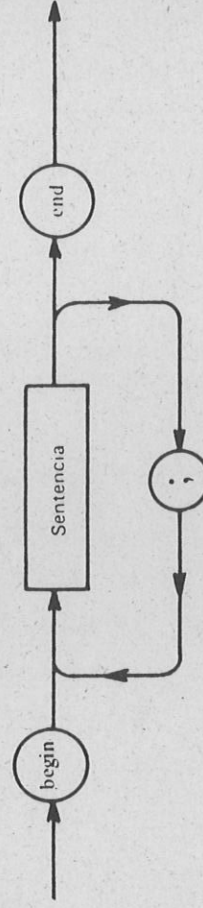
Sección de parámetros formales



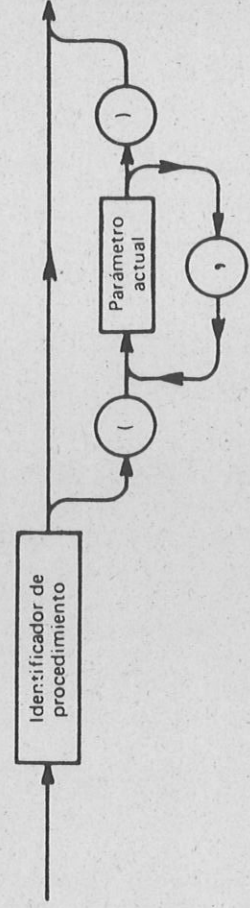
Sentencia



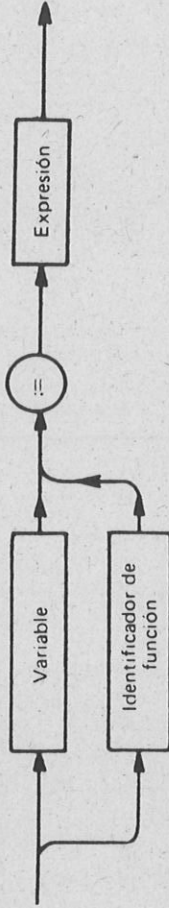
Sentencia compuesta



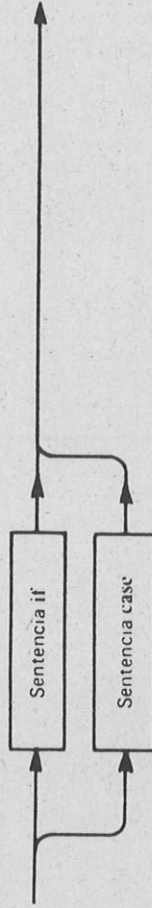
Sentencia o llamada de procedimiento



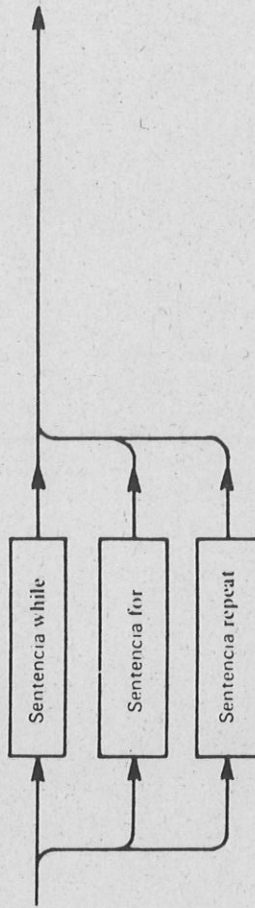
Sentencia de asignación



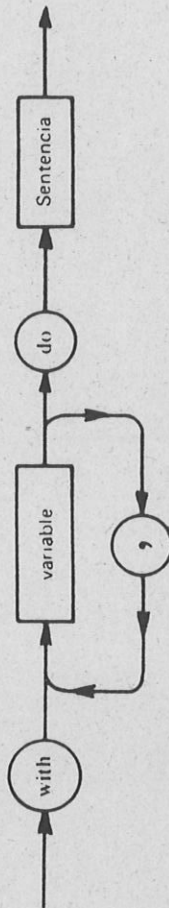
Sentencia selectiva



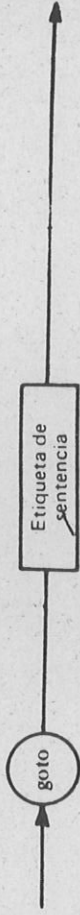
Sentencia repetitiva



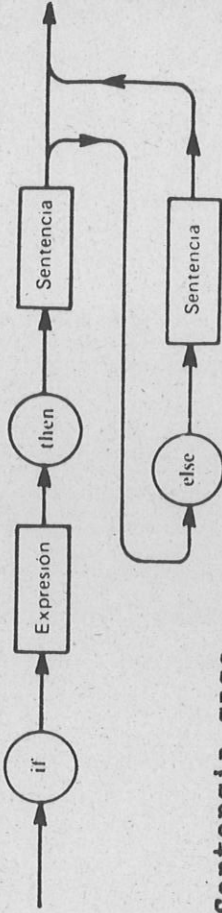
Sentencia with



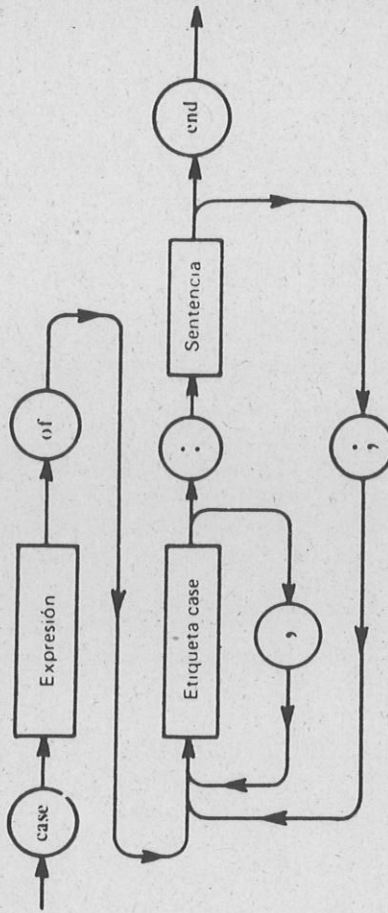
Sentencia goto



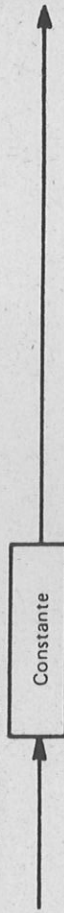
Sentencia if



Sentencia case

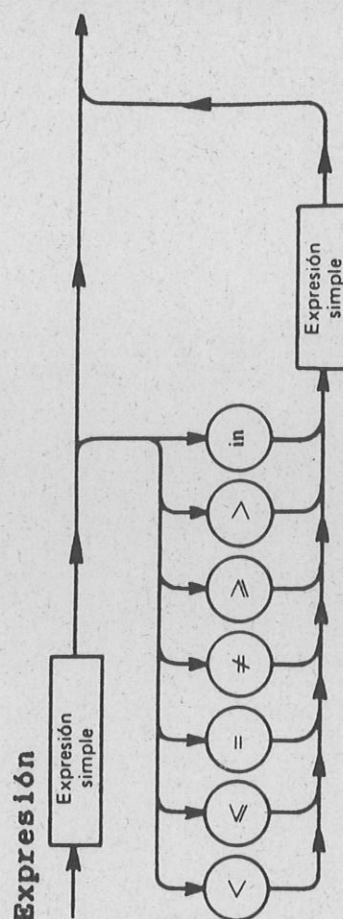
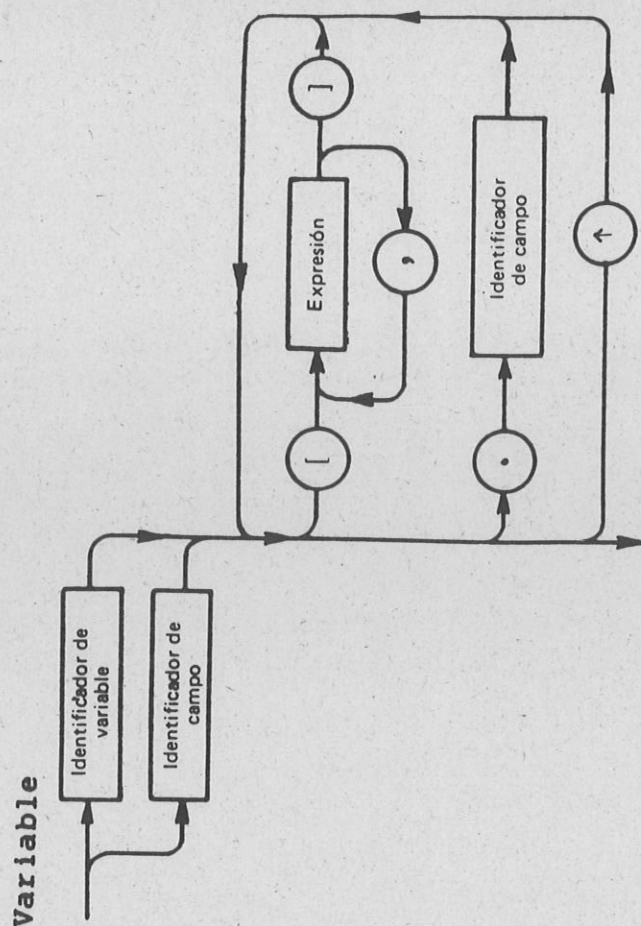
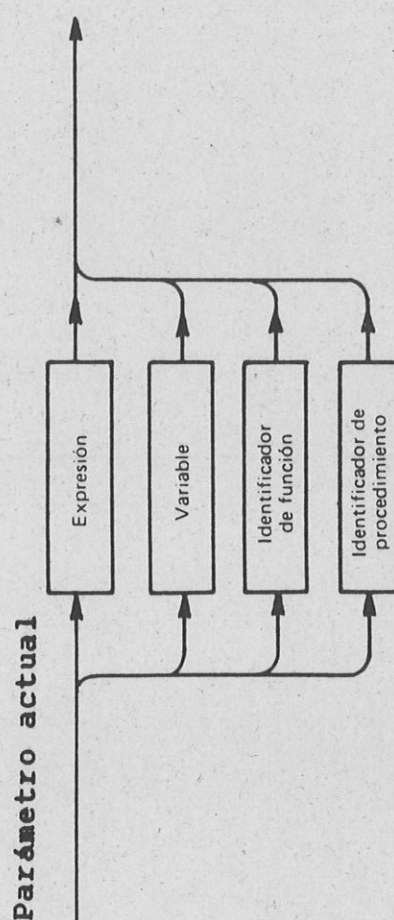
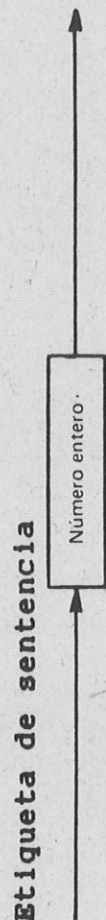
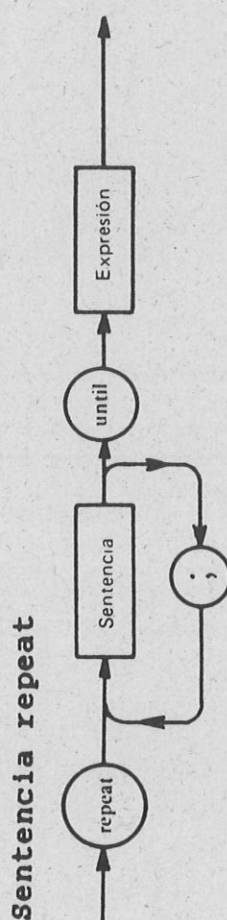
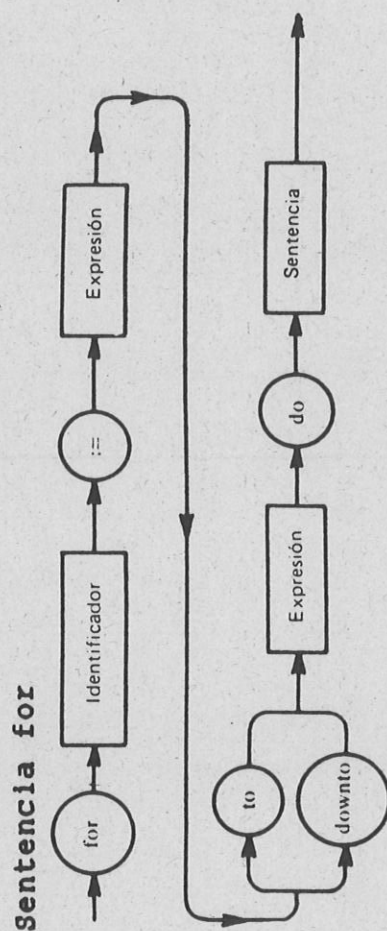


Etiqueta case

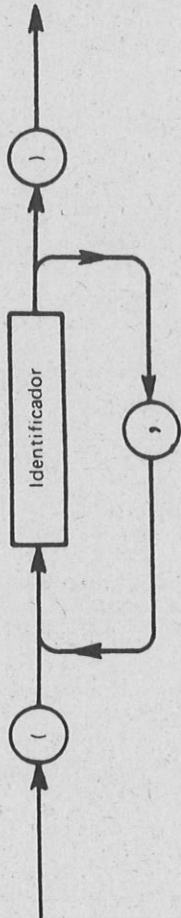


Sentencia while





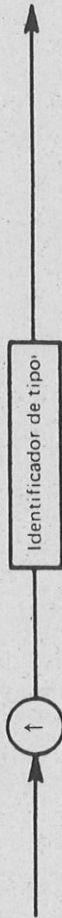
Tipo enumerado



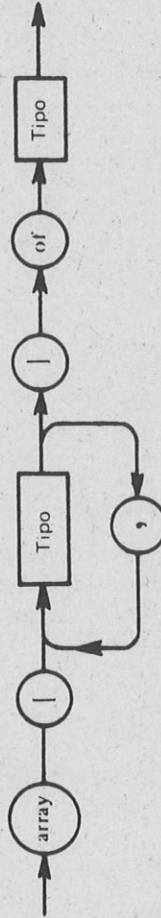
Tipo subrango



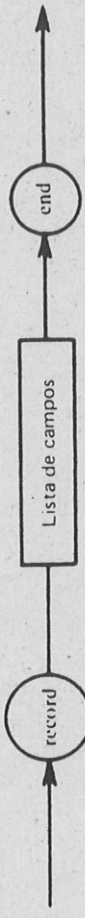
Tipo apuntador



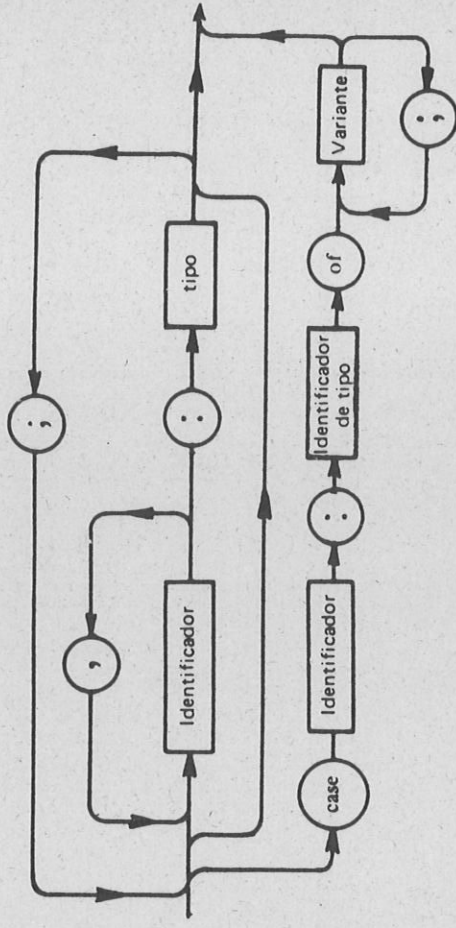
Tipo array



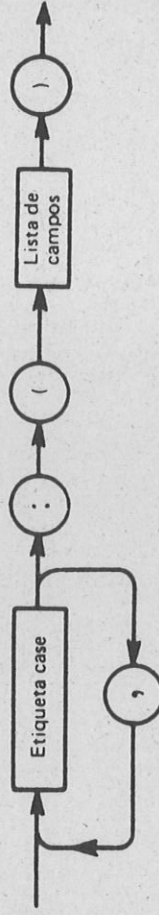
Tipo registro



Lista de campos



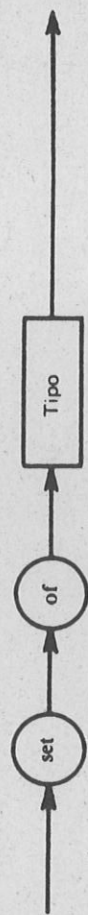
Variante



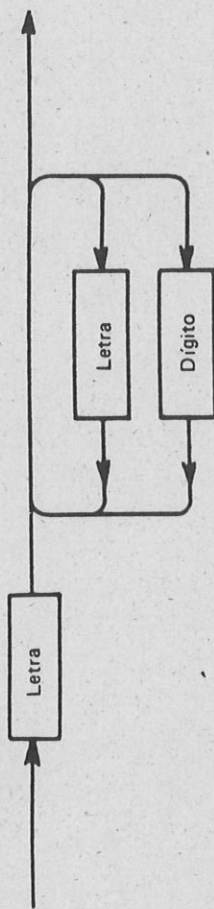
Tipo fichero



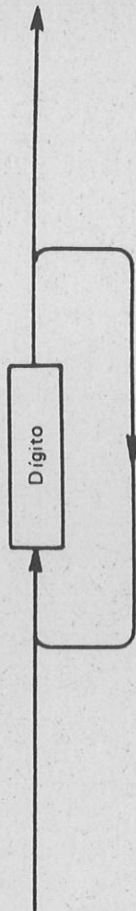
Tipo conjunto



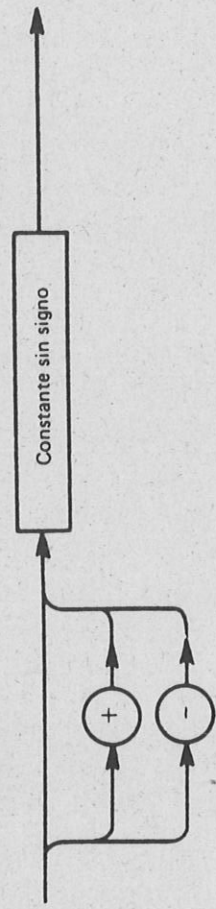
Identificador



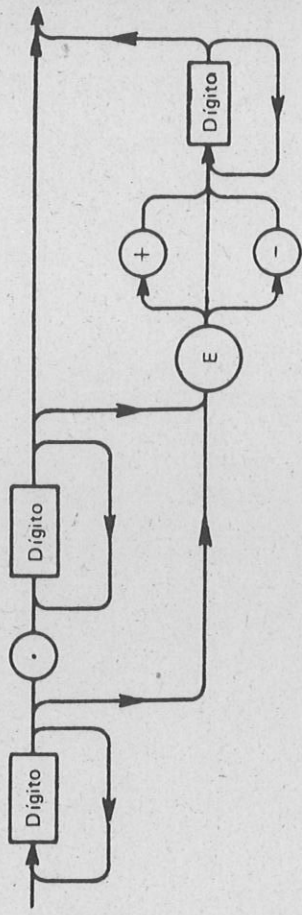
Número entero



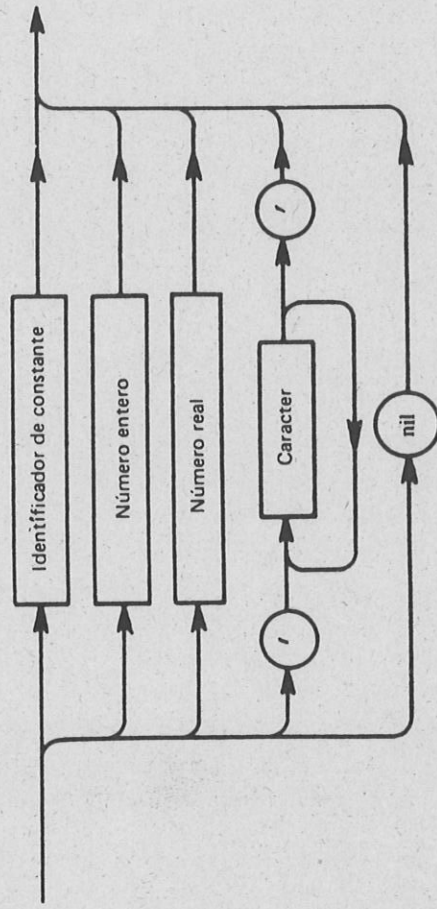
Constante



Número real



Constante sin signo



Palabras reservadas

AND	END	MOD	REPEAT
ARRAY	FILE	NIL	SET
BEGIN	FOR	NOT	THEN
CASE	FORWARD	OF	TO
CONST	FUNCTION	OR	TYPE
DIV	GOTO	PACKED	UNTIL
DO	IF	PROCEDURE	VAR
DOWNTO	IN	PROGRAM	WHILE
ELSE	LABEL	RECORD	WITH

Identificadores estándares

Constantes estándares

False True MaxInt

Tipos estándares

Integer Boolean Real Char Text

Ficheros estándares

Input Output

Funciones estándares

	Tipo del parámetro	Tipo del resultado	Devuelve
Abs(X)	Integer o Real	Lo mismo como parámetro	Valor absoluto de X
ArcTan(X)	Integer o Real	Real	Arco tangente de X en radianes
Chr(X)	Integer	Char	Carácter cuyo número ordinal es X
Cos(X)	Integer o Real	Real	Conseno de X (X en radianes)
EOF(F)	File	Boolean	Test sobre fin del fichero F
EOLN(F)	File	Boolean	Test sobre fin del fichero F e a la potencia X
Exp(X)	Real o Integer	Real	Logaritmo natural de X
Ln(X)	Real o Integer	Real	Test sobre si X es impar
Odd(X)	Integer	Boolean	Número ordinal de X
Ord(X)	Ordinal (escalar excepto el Real)	Integer	
Pred(X)	Ordinal (escalar excepto el Real)	Lo mismo como parámetro	El único predecesor de X (excepto cuando X es el primer valor)
Round(X)	Real	Integer	Redondeo de X
Sin(X)	Real o Integer	Real	Seno de X (X en radianes)
Sqr(X)	Real o Integer	Lo mismo como parámetro	Cuadrado de X
Sqrt(X)	Real o Integer	Real	Raíz cuadrada de X
Succ(X)	Ordinal (escalar excepto el Real)	Lo mismo como parámetro	El único sucesor de X (excepto cuando X es el primer valor)
Trunc(X)	Real	Integer	X truncado

Procedimientos estándares

Descripción

Dispose(P)	Destruye la variable dinámica referenciada por el puntero P devolviéndola a la lista de memoria disponible.
Get(F)	Avanza la posición actual del archivo F a la siguiente componente y le asigna el valor de la componente a F [^] .
New(P)	Crea una variable de tipo referenciado por el puntero P y almacena un puntero a la nueva variable P.
Pack(U, I, P)	Copia los elementos del array U comenzando en la posición del subíndice I en el array empaquetado P comenzando en la posición del primer subíndice.
Page(F)	Avanza hasta el comienzo de la siguiente página antes de imprimir la siguiente línea del archivo de texto F.
Put(F)	Añade el valor de la variable F [^] al archivo F.
Read(F, lista de variables)	Lee datos del archivo de texto F (si no se especifica F, se toma INPUT por defecto) y le asigna estos valores en orden a las variables de la lista de variables hasta que se completa la lista.
Readln(F, lista de variables)	Lo mismo que READ excepto que el puntero al archivo se avanza hasta pasar el fin de línea, después de que se haya completado la lista de variables.

Reset(F)

Rewrite(F)

Unpack(F, U, I)

Write(F, lista de parámetros)

WriteIn(F, lista de parámetros)

Prepara el archivo F para comenzar la lectura.

Prepara el archivo F para comenzar la escritura en él, se pierde el contenido anterior de F.

Copia los elementos del array empaquetado P comenzando en la posición del primer subíndice en el array U comenzando en la posición del subíndice I.

Escribe los datos en el archivo de texto F en el orden especificado por la lista de parámetros (si no se especifica F, se escribe por defecto en el fichero OUTPUT).

Lo mismo que WRITE excepto que genera un fin de línea después de completar la lista de parámetros.

(Algunos compiladores suministran algunos tipos, archivos, funciones y/o procedimientos adicionales. Consulte en el manual de la implementación concreta que utilice lo que está disponible.)

Operadores y símbolos del Pascal

Símbolos estándares	Símbolos alternos
+	más o unión de conjuntos
-	menos o diferencia entre conjuntos
*	multiplicación o intersección de conjuntos
/	división
DIV	división entera
MOD	resto de la división entera (módulo)
<	es menor que
<=	es menor o igual que
=	es igual a
<>	no es igual a
>	es mayor o igual que
>=	es mayor que
>	conjunción Booleana
AND	disyunción inclusiva Booleana
OR	negación Booleana
NOT	es miembro de un conjunto
IN	asignación
=	separa los elementos de una lista
.	separa sentencias
:	separa el nombre y el tipo de las variables;
:	separa los rótulos de la sentencia case y las sentencias;
:	separa el rótulo de una sentencia de caracteres
:	delimita el carácter y las cadenas de caracteres
:	punto decimal, selector de registro y terminación del programa
:	especifica el subrango
:	indicador de variable puntero y fichero
:	comienzo de lista de parámetros o de expresión anidada
:	fin de lista de parámetros o de expresión anidada
:	comienzo de lista de subíndices o de expresión de conjunto
:	fin de un comentario
:	fin de lista de subíndices o de expresión de conjunto

Precedencia de operadores

NOTA:

- Los paréntesis pueden usarse para cambiar el orden de precedencia.
- Cuando se usan operadores de la misma precedencia, se ejecutan de izquierda a derecha.

NOT

Precedencia más alta

* / DIV MOD AND

+ - OR

< <= = >= > <> IN Precedencia más baja

Conjunto de caracteres

La siguiente tabla muestra la ordenación de los conjuntos de caracteres más usados: ASCII (American Standard Code for Information Interchange).

Sólo se permiten los caracteres imprimibles. El número ordinal de cada carácter se muestra en decimal. El carácter blanco se denota por un "0".

Digito(s) de la izquierda		Digito de la derecha	ASCII																
3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9
(	2	<	F	P	Z	d	n	x		)	3	=	G	I	e	o	Y		
										[	4	>	H	R	/	f	p	z	
										-	+	5	?	I	S	1	g	q	!
										.	6	@	J	T	^	h	r	-	
										#	-	7	A	K	U	_	i	s	!
										\$	8	B	L	V	,	i	t	.	
										%	/	9	C	M	W	a	k	u	
										&	0	.	D	N	X	b	-	v	
										.	1	.	E	O	Y	c	e	w	